Université d'Orléans
Laboratoire d'Informatique Fondamentale d'Orléans

Barbara FILA–KORDY

A tomates po r l'analyse de doc ments XML compressés, applications à la séc rité d'accès

Thèse dirigée par : Professeur Siva ANANTHARAMAN

JURY :
Professeur Françoise GIRE, Université de PARIS 1
Professeur Sophie TISON, Université de LILLE 1
Professeur Siva ANANTHARAMAN, Université d'ORLÉANS
Professeur Michael BENEDIKT, Universit of OXFORD
Directeur de Recherche Michaël RUSINOWITCH, INRIA–Lorraine
Professeur Moshe VARDI, RICE Universit

Thèse soutenue le 4 novembre 2008 à Orléans

Table des matières

Introduction

Le problème de l'extraction d'information dans des documents semi–structurés, du type XML, constitue un des plus importants domaines de la recherche actuelle en informatique. Il a généré un grand nombre de travaux tant d'un point de vue pratique, que d'un point de vue théorique (voir Chapitre 1). Dans ce travail de thèse, nous nous sommes fixés deux objectifs :

1. évaluation des requêtes sur un document assujetti à une politique du contrôle d'accès,
2. évaluation des requêtes sur un document pouvant être partiellement ou totalement compressé.

Notre étude porte essentiellement sur l'évaluation des requêtes unaires, c.–à–d., sélectionnant un ensemble des nœuds du document, qui satisfont les propriétés spécifiées par la requête. Pour exprimer les requêtes, nous utilisons XPath [92] — le principal langage de sélection dans les documents XML. Grâce à ses axes "navigationels", et ses filtres qualificatifs, XPath permet la navigation dans des documents XML, et la sélection des nœuds répondant à la requête.

XML (eXtensible Merkup Language) [9] est devenu le standard de représentation et d'échange des données sur le WEB. Il est basé sur un simple mais puissant concept, celui des éléments balisés. Un élément balisé peut représenter une petite partie du document, ou un objet très complexe. Les éléments peuvent être imbriqués, c.–à–d., un élément peut être composé d'autres éléments. De plus, des attributs qui apportent des informations supplémentaires, peuvent être associés aux éléments. Au Chapitre 2 nous présentons brièvement le cadre de notre travail : les documents XML et le langage XPath. Nous y définissons également deux fragments de XPath, auxquels nous nous intéresserons par la suite.

L'utilisation très répandu d'XML a entraîné la nécessité d'introduire des modèles et techniques pour sécuriser les données XML. La sécurisation est cruciale pour faciliter la propagation des données qui contiennent des informations représentant les différents niveaux de sensibilité et d'accessibilité. Dans ce travail nous nous focalisons sur les aspects de sécurité liés à l'autorisation d'accès à l'information — la confidentialité. Cette dernière est en général assurée par des mécanismes de contrôle d'accès (politiques du contrôle d'accès, clés etc.). Nous avons choisi de modéliser les politiques du contrôle d'accès par des clauses du premier ordre, et plus spécifiquement par des clauses de Horn, pouvant être soumises à des contraintes. Un tel choix nous permet d'évaluer des requêtes sur les documents XML, en utilisant une approche basée sur des systèmes de transitions appropriés (Chapitre 3). L'idée est d'exprimer les transitions de ces systèmes également par des clauses de Horn contraintes, et

ensuite de les coupler avec celles traduisant la politique du contrôle d'accès. Il n'est pas difficile de montrer qu'avec une telle modélisation on couvre bien d'autres approches, telles que par exemple RBAC (Role Based Access Control), ou celles basées sur l'attribution de clés aux nœuds et/ou aux attributs.

Tout document XML a une structure arborescente, donc d'une manière naturelle, est représenté par un arbre. Néanmoins, une telle représentation est souvent redondante, car sur un arbre, la même information peut figurer plusieurs fois. L'utilisation des structures compressées – telles que les dags (directed acyclic graphs), ou les grammaires d'arbres appropriées – au lieu des arbres, permet alors d'optimiser considérablement l'espace de stockage des documents, ainsi que le temps d'évaluation des requêtes. Les avantages d'utilisation des structures XML compressées ont été étudiés, entre autres, dans [35, 16, 64].

Les documents que nous considérons dans ce travail peuvent être totalement ou partiellement compressés. Un document est donné sous une forme totalement compressée, si le dag qui le représente contient toute information une seule fois. Toute représentation intermédiaire, entre l'arbre et la forme totalement compressée d'un document donné, est une forme partiellement compressée de ce dernier. Au Chapitre 4, nous développons une approche pour l'évaluation des requêtes positives de Core XPath, sur les documents compressés représentés par les dags. Elle est basée sur sept automates de mots, correspondant aux sept axes de base de Core XPath. Grâce à une définition appropriée, ces automates peuvent courir d'une façon descendante, sur les dags. Cette méthode nous permet d'évaluer des requêtes sur les documents partiellement ou totalement compressées, sans devoir décompresser ces derniers. De plus, pour une requête Q et un document (compressé) t donnés, l'évaluation de Q sur t peut être adaptée de façon à fournir exactement la même réponse que l'évaluation de Q sur l'arbre représentant t (Section 4.).

Il importe de noter que la complexité des approches que nous avons développées, reste comparable à celle des méthodes connues : linéaire par rapport au nombre d'arêtes du document donné, et par rapport à la taille de la requête considérée. En particulier, elle est linéaire par rapport au nombre de nœuds, si le document est arborescent (non–compressé).

Dans la dernière partie de notre travail, nous esquissons une approche qui traite du problème d'inclusion des schémas de requêtes (pattern containment, [6 ,]). Cette approche est basée sur des techniques de réécriture. On remarque que tout document XML étant un arbre, peut être vu comme un "pattern" (au sens de [6]). De plus, toute requête positive de Core Xpath, n'utilisant que des axes `child` et `descendant`, définit aussi un tel pattern. Résoudre le problème d'inclusion de patterns revient de fait à répondre aux questions suivantes :
soient deux requêtes Q, Q' et un document t :

- vérifier si toute réponse à Q constitue également une réponse à Q' ($Q \subseteq Q'$);
- vérifier si Q et Q' ont exactement les même réponses sur tout document ($Q \equiv Q'$);
- vérifier si Q admet (au moins) une réponse sur t ($t \subseteq Q$).

Au Chapitre 5, nous définissons un système de règles de réécriture basées sur la sémantique de l'inclusion de patterns, et montrons que pour deux patterns donnés Q et Q', Q est inclus dans Q' si et seulement si, il est possible de réécrire Q vers

Q', en utilisant les règles de ce système. Il est important de noter que cette vision de requêtes comme des patterns peut servir de base pour le traitement des requêtes n–aires, c.–à–d., celles qui sélectionnent un ensemble des n–uplets de nœuds. A noter également que les résultats sur l'inclusion de patterns, obtenus dans ce travail, restent valides même lorsque les modèles des patterns sont des documents compressés.

Chapitre 1

État de l'art

Cet état de l'art peut être vu comme le point de départ pour la présente thèse. Nous y énonçons les plus importants résultats connus, concernant l'évaluation des requêtes sur les documents XML (Section 1.1), le contrôle d'accès (Section 1.2), la compression (Section 1.3), et le problème d'inclusion et de minimisation des requêtes (Section 1.4). Nous esquissons brièvement les différentes méthodes existantes, ainsi que leurs complexités. Nous essayons également de situer nos travaux (détaillés dans la suite de ce rapport) par rapport à ces résultats connus.

1.1. Évaluation de requêtes sur documents XML

XML, standardisé en janvier 1998, joue un rôle de plus en plus important pratiquement dans toutes les branches d'informatique contemporaine. Concevoir les outils efficaces pour l'extraction des informations dans les documents XML est devenu l'objectif principal de nombreux chercheurs travaillant dans le domaine des bases de données. Plusieurs fragments des différentes logiques (FOL [10, 9], logique modale [64], MSO [5], datalog monadique [35, 40] etc.) ont été étudiés dans le contexte d'évaluation des requêtes sur les documents XML. MSO (Monadic Second–Order Logic) a été proposé dans [5] comme un repère pour le pouvoir d'expression des langages de sélection. Néanmoins, MSO n'est pas approprié pour constituer un langage de requêtes pratique, car il permet d'exprimer des requêtes très complexes d'une façon très concise, ce qui entraîne que le problème d'évaluation devient intraitable [34]. Or, il existe des langages — comme le μ-calcul modal ou le datalog monadique [35, 40] — qui ont le même pouvoir d'expression sur les arbres que MSO, mais où l'évaluation de requêtes est plus efficace. Par exemple, la complexité d'évaluation des requêtes de datalog monadique sur les documents XML, est montrée linéaire par rapport à la taille du programme datalog correspondant, et la taille du document considéré [39].

L'utilisation de plus en plus répandu de XML a suscité, durant la dernière décennie, le développement des langages de requêtes, tels que XPath, XQuery, spécialement conçus pour traiter les documents XML. XPath [92] est un langage pour adresser certaines parties des documents XML. Il est également incorporé dans plusieurs formalismes liés à XML, tels que :

- XSLT [93, 15] (eXtensible Style sheet Language Transformations) — un langage permettant de transformer un document XML en un autre document XML;

- XQuery [99] — un langage de requêtes d'ordre supérieur, qui offre entre autres, une possibilité de modifier le résultat d'évaluation d'une requête, et de produire des nouveaux documents XML;
- XPointer [96] — une spécification du W3C, dont l'objectif est de permettre de référencer un fragment d'un document XML externe en ligne;
- XML Schema [95] et XLink [94] — où des expressions XPath servent à définir des clés d'accès.

XPath Version 1.0 à été publié comme une recommandation de W3C en 1999. C'est un langage sans variables, conçu pour sélectionner des nœuds dans les documents XML, en spécifiant des chemins d'accès. De tels chemins peuvent alternativement être formulés dans la logique de premier ordre (FOL). Néanmoins, la partie de navigation de XPath, dite Core XPath, n'est pas assez puissante pour exprimer tous les chemins d'accès définissables en FOL. Michael Benedikt et Christoph Koch montrent dans [10] que le Core XPath couvre précisément toutes les requêtes de FOL avec deux variables sur la structure de navigation de documents XML. Dans [63, 65], Maarten Marx introduit XPath avec des axes conditionnels (en anglais *Conditional XPath*), une extension de Core XPath (toujours sans variables), dans laquelle il est possible d'exprimer tout chemin d'accès sur un arbre XML, définissable en FOL. La toute dernière version de XPath [98] (XPath 2.0, recommandée par W3C depuis 200), étend considérablement le pouvoir d'expression de XPath 1.0. Elle contient des variables, et offre une possibilité d'exprimer des formules quantifiées de FOL. Par conséquent, elle couvre pratiquement tout FOL.

Dans le présent travail, nous nous restreignons à XPath 1.0, qui est toujours largement employé par les utilisateurs d'XML. Nous présentons ici deux approches permettant d'évaluer des requêtes positives de XPath, sur les documents XML : la première (Chapitre 3) est basée sur des systèmes de transitions, et la deuxième (Chapitre 4) sur des automates de mots. La théorie des automates a été largement utilisée pour l'évaluation de requêtes sur les documents XML [5, 4, 6, 39, 3, 44, 19, 62, 8]. Dans le contexte étudié ici, les plus importants sont des automates de requêtes, proposés par Neven et Schwentick dans [6], pour l'évaluation des requêtes MSO sur les arbres d'arité fixe (en anglais *ranked*) et non fixe (*unranked*). Un automate de requête est un automate déterministe bi–directionnel d'arbres, qui a une capacité de sélectionner des nœuds, grâce à un ensemble d'états sélectionnants. Nous utilisons ce concept d'états sélectionnants dans notre approche présentée dans le Chapitre 4.

Les arbres d'arité non fixe constituent les meilleurs modèles pour les documents XML, mais leur utilisation peut être problématique, car ils peuvent être récursifs en largeur et en profondeur. Pour aborder ce problème plusieurs méthodes on été employées : Dans [41] Gottlob et al. redéfinissent tous les axes de XPath en n'utilisant que deux relations *firstchild* et *nextsibling*, ce qui permet d'évaluer des requêtes sur une représentation binaire (classique du type *first–child next–sibling*) d'un document — arbre d'arité non fixe — donné. Des *automates de haie* (en anglais *hedge automata*) [49, 2] utilisent un niveau de récursivité supplémentaire dans des transitions, pour exprimer la récursivité horizontale. Pourtant, une telle extension syntaxique entraîne de nombreux problèmes techniques [39, 6]. Des automates

d'arbres *stepwise*, définis dans [19], courent sur une nouvelle représentation binaire d'arbres d'arité non fixe. En utilisant ce codage, les auteurs de [19] montrent que les automates d'arbres stepwise ont le même pouvoir d'expression que le datalog monadique et MSO. De plus, il a été montré dans [62] que ces automates fournissent des représentations très concises, ce qui est important de point de vue du problème de minimisation.

La question de la complexité d'évaluation des requêtes XPath, sur les documents XML, a été abordée dans [41, 42] pour les documents arborescents, et dans [16, 35] pour les documents compressés (pour plus d'information sur l'évaluation des requêtes XPath sur les documents XML compressés, voir la Section 1.3). En se basant sur des observations expérimentales, les auteurs de [41] constatent que l'évaluation d'une requête Q donnée nécessite (dans le pire de cas), dans la majorité des évaluateurs de XPath existants, un temps exponentiel par rapport à la taille de Q. Pourtant, des requêtes XPath peuvent être évaluées beaucoup plus efficacement. Gottlob, Koch et Pichler proposent dans [41] des algorithmes qui permettent d'évaluer les requêtes quelconques de XPath en temps polynomial combiné, c.–à–d., par rapport à la taille de la requête et la taille du document. D'ailleurs, ils montrent comment implémenter leurs algorithmes dans des évaluateurs de XPath existants. Ils distinguent deux fragments de XPath — Core XPath et XPatterns — pour lesquels il existe des algorithmes linéaires (en temps) d'évaluation. Dans la version étendue de leur article ([43]), Gottlob, Koch et Pichler développent les résultats précédents, en prouvant que la complexité combinée d'évaluation de requêtes XPath est PTIME–hard. Ils identifient également plusieurs fragments de XPath, fréquemment utilisés, pour lesquels la complexité combinée du problème d'évaluation est dans une classe de complexité parallélisable NC^2. Pour finir cette section, notons que la complexité en temps de nos deux approches d'évaluation de requêtes, présentées dans les Chapitres 3 et 4, est linéaire par rapport à la taille de la requête et le nombre d'arêtes du document considéré.

1.2. Contrôle d'accès aux documents XML

Puisque les systèmes informatiques fournissent des applications multiples aux utilisateurs multiples, la protection des données a toujours été une question clé dans le contexte de systèmes d'information. Le rôle du contrôle d'accès est de permettre ou interdire à des sujets (utilisateurs ou processus), d'exécuter des opérations (lire, écrire, modifier, supprimer etc.) sur des objets (données ou programmes) dans le système informatique [5]. Grâce au contrôle d'accès, le système peut limiter l'accès à certaines données, aux utilisateurs non autorisés. Ainsi, les bases de données relationnelles sont en général accompagnées d'un mécanisme de contrôle d'accès, intégré dans le système, et basé sur des vues, des droits d'accès, ou des tables de privilèges accordés aux utilisateurs [20].

L'utilisation très répandu de XML a entraîné la nécessité d'introduire des modèles et techniques pour sécuriser les données XML. Un certain nombre de normes (comme OASIS standard [9], ou XACL [51]) ont été introduites, pour aborder le problème du contrôle d'accès aux documents XML. Différentes approches [1, 3 , 26, 2 , 14,

28, 2, 5] montrent que le contrôle d'accès, dans le contexte de documents XML, n'est pas un sujet trivial. Il en est ainsi pour plusieurs raisons :

- la nature *semi–structurée* — la structure d'un document XML n'est pas forcément connue à l'avance, comme dans le cas de tables relationnelles — un document XML n'est pas toujours accompagné d'un DTD ou d'un schéma;
- la nature *descendante* — dans un document XML, la sémantique de chaque nœud dépend souvent de celle de ses ancêtres, par suite, dans plusieurs approches de sécurisation d'XML, on suppose que si l'accès à un nœud est refusé, alors il faut interdire aussi l'accès a tous ses descendants;
- la nature *hiérarchique* — il est très utile, dans le contexte de XML, de spécifier une politique du contrôle d'accès, qui ne peut être applicable à un nœud que si ce dernier satisfait une condition donnée (par exemple, si un attribut donné admet au nœud en question une valeur spécifique).

Comme le mentionnent les auteurs de [13], la sécurisation de XML comporte trois problèmes : *confidentialité*, *intégrité* et *authenticité*. La confidentialité garantit que l'information donnée ne peut être accessible que pour un utilisateur autorisé, conformément aux politiques d'accès spécifiées. L'intégrité et l'authenticité sont liées à la diffusion et l'échange des données : la première assure que la transmission des données depuis la source vers le destinataire n'entraîne pas de changement du contenu d'information, tandis que la seconde garantit au destinataire recevant des données que ces dernières viennent bien de la source mentionnée. La confidentialité est en général assurée par les mécanismes du contrôle d'accès (politiques du contrôle d'accès, clés etc.). Pour garantir l'intégrité, on utilise les mécanismes du contrôle d'accès ainsi que les techniques de cryptage. L'authenticité peut être assurée, par exemple, par les techniques utilisant les signatures numériques [8].

Dans la présente thèse nous nous concentrons sur la confidentialité. D'une manière générale, on peut définir une *politique du contrôle d'accès* comme un ensemble de règles [36]. Chacune de ces *règles de contrôle d'accès* est un uplèt de la forme

$$rule = (user, data, action, function, scope),$$

où :

- *user* précise quel utilisateur est concerné par *rule*,
- *data* est la donnée que *user* est/n'est pas autorisé à accéder,
- *action* définit une action (lire, écrire, modifier, supprimer etc.) que *user* est/n'est pas autorisé à exécuter sur *data*,
- *function* spécifie si *rule* autorise ou pas *user* à effectuer *action* sur *data*,
- *scope* détermine la portée de *rule* (la valeur d'un attribut à un nœud, un nœud avec tous ses couples attribut=valeur, un nœud et ses descendants etc.).

Considérons une règle de contrôle d'accès $rule = (user, data, action, function, scope)$. Souvent on suppose que certains paramètres de *rule* sont fixés; par exemple les auteurs de [36] fixent les valeurs de *user* et *action*, et ne focalisent que sur les règles du type : l'information définie par *data* et *scope* est ou n'est pas visible. La nature hiérarchique des documents XML impose qu'il faut définir la portée (*scope*) de chaque règle de contrôle d'accès. Dans la plupart des méthodes proposées [14, 3 , 51, 9], *scope* est limité à un nœud. Les autres unités de protection peuvent être : un

nœud et ses attributs (comme dans [2]), un nœud avec son nœud enfant du type *text* ([1]), un nœud avec tous ses attributs ainsi que tous ses descendants et leurs attributs ([3 , 51, 14, 2 , 1]). Il est possible que la portée d'une règle couvre seulement des descendants d'un nœud, qui se trouvent à une certaine profondeur dans l'arbre considéré (voir [14]). Toute règle de contrôle d'accès, dont la portée n'est qu'un seul nœud et éventuellement ses attributs, est appelée *locale*. Chaque règle où la portée définit un nœud avec tous ses descendants, est dite *récursive* ([36]). Il est très commode de définir les composantes $function$ et *scope* à l'aide des labels supplémentaires. Ainsi, les approches présentées dans [2] et [66], sont basées sur l'attribution des clés aux nœuds ou aux attributs, qui autorisent/interdisent l'accès à l'information stockée. Les auteurs de [1] utilisent quatre labels $+r$, $+R$, $-r$ et $-R$, pour spécifier la possibilité d'accéder à un nœud et ses descendants. Soit un document donné t, et un nœud u de t :

- si u est labelé par $+r$, alors l'information stockée à u est accessible,
- si u est labalé par $+R$, alors les informations stockées à u et à ses descendants sont accessibles,
- si u est labelé par $-r$, alors l'information stockée à u n'est pas accessible,
- si u est labalé par $-R$, alors les informations stockées à u et à ses descendants ne sont pas accessibles.

Les différentes logiques (FOL, LTL etc.) sont également employées, pour définir des politiques du contrôle d'accès ([1], [11], [46]). Dans [1] Martin Abadi présente une discussion sur le rôle de la logique dans le contrôle d'accès. Dans [11] les auteurs introduisent la notion de *non–interférence* d'une requête Q avec un ensemble d'éléments X d'un document donné t, qui garantit que l'évaluation de Q sur t fournit toujours le même résultat, même si on change le contenu d'éléments de X (par exemple, les valeurs de certains attributs). Par suite, ils définissent des requêtes dites *sûres* (en anglais *safe*), dont l'évaluation n'entraîne pas de violation des politiques du contrôle d'accès données. Ces dernières, formulées dans [11] à l'aide de la logique des propositions, expriment des contraintes qui doivent être satisfaites par le résultat d'évaluation de Q sur t, pour pouvoir fournir ce résultat à l'utilisateur. Dans [46] Pieter Hartel propose d'utiliser la logique LTL, pour garder la trace de tous les nœuds intermédiaires auxquels le moteur d'évaluation a accédé pendant l'évaluation d'une requête. Ceci permet une analyse détaillée de tout renseignement lié à la requête en question. La trace peut contenir des informations sensibles ou interdites, qui ne sont pas visibles explicitement dans le résultat d'évaluation. Cette trace est ensuite utilisée pour garantir un accès sûr et sécurisé à l'information autorisée.

Dans la Section 3.6 nous présentons une approche qui permet d'évaluer des requêtes positives de XPath sur les documents assujettis à des politiques du contrôle d'accès. Comme dans [36], nous supposons que l'action définie par toutes les règles de contrôle d'accès est fixée (il s'agit de la possibilité de voir l'information définie par *data*). Nos règles de contrôle d'accès sont formulées en termes de clauses de la logique de premier ordre, plus précisément des clauses de Horn avec contraintes qui spécifient leur portée. Nous gardons — sous forme des clauses — la trace de toute information à laquelle chaque utilisateur (ou groupe d'utilisateurs) a eu l'accès jusqu'au moment présent. Par la suite, la résolution clausale est employée, pour garantir à un

utilisateur donné, l'accès à l'information à laquelle il a le droit d'accéder. Dans la Section 3. , on montre que notre vue clausale est appropriée pour encoder la plupart des approches que nous avons mentionnées.

1.3. Compression

Une augmentation massive des volumes de données a motivé récemment un intérêt pour l'utilisation des structures de données compressées. Le but principal est de développer des algorithmes, des approches et des outils qui travailleraient directement sur les données compressés, sans nécessiter une décompression. Une telle vue permet d'économiser l'espace de stockage et le temps nécessaire pour le traitement des documents. La compression offre une possibilité de stocker des documents volumineux dans la mémoire principale, où ils peuvent être accédés plus facilement, et manipulés plus rapidement (mises à jours, évaluation de requêtes).

Dans un premier temps, beaucoup de chercheurs se sont intéressés aux problèmes de compression des textes unidimensionnelles (mots) et 2–dimensionnelles [58, 29, 81, 56, 83, 12]. Parmi les principales questions étudiées, on peut citer les suivantes :

- *querying problem* [56] — quel est le symbole à une position donnée dans un mot représenté sous une forme compressée?
- *compressed pattern matching problem* [80, 6] — rechercher un texte donné dans un autre texte compressé donné;
- *fully compressed pattern matching problem* [38, 0] — rechercher un texte compressé donné dans un autre texte compressé donné;
- *(fully) compressed embedding problem* [56] — est–ce qu'un text (compressé) peut être plongé dans un autre texte représenté sous une forme compressée?
- *longest and shortest common subsequence compressed problems* — trouver le plus long/court sous–mot commun dans un ensemble des mots compressés (ces problèmes ont de très nombreuses applications, p.ex. en biologie informatique [45]).

Différents algorithmes de compression de textes ont été développés. Les plus connus sont : straightline programs (SLP — grammaires hors contexte de mots générant un seul mot) [80, 83], et Lempel–Ziv factorisations (LZ , LZ 8 — compressions basées sur le principe de remplacer des sous–mots d'un mot considéré, par des pointeurs indiquant leurs occurrences précédentes) [101, 102, 84]. Les recherches théoriques ont également conduit à l'élaboration de nombreux compresseurs de texte, très puissants, et indispensables à nos jours (ZIP, GZIP, BZIP2, PKZIP, ARJ etc.).

Durant la dernière décennie, c'est la compression des structures plus complexes (telles que arbres et images), qui attire plus d'attention [59, 61, 59, 1 , 16, 30, 35, 60, 4]. L'arbre est une structure de données fondamentale utilisée dans plusieurs branches d'informatique, telles que la réécriture, le model checking, les bases de données etc. La façon la plus naturelle de compresser un arbre est de le représenter sous forme d'un graphe orienté sans cycles (*dag* — directed acyclic graph), en suivant le principe de *partager des sous–arbres communs*. Une telle représentation préserve la structure du document d'origine, et peut être encodée par une grammaire régulière et straightline d'arbres [61]. L'idée d'utiliser la structure de dags à la place des arbres, pour les documents XML, a été largement développée dans [16, 64] et [35]. Ces

travaux montrent que des requêtes peuvent être efficacement évaluées directement sur les dags, et que le résultat d'une telle évaluation peut être également fourni en forme comprimée.

Les automates d'arbres ont été largement utilisés pour évaluer les requêtes sur les documents XML arborescents [40, 39, 4, 6]. Dans le cas où le document est donné sous une forme compressée, il est donc légitime d'essayer d'étendre/adapter ces approches aux automates de dags. La notion d'automate de dag (DA) est définie dans [22], comme une extension naturelle d'un automate d'arbre bottom–up. Charatonik montre dans [22] que le problème d'appartenance (membership) pour ces automates est dans NP, et que celui du vide (emptiness) est NP–complet. Mais, les auteurs de [ ] prouvent que l'ensemble de tous les arbres représentés par un ensemble de dags acceptés par un DA non–déterministe ne forme pas toujours un langage régulier d'arbres. Cela implique que la classe d'arbres pouvant être reconnus par les DAs, est une superclasse stricte de la classe des langages d'arbres réguliers. Il en résulte que l'utilisation des DAs, tels que définis dans [22,], pour l'évaluation des requêtes sur les documents compressés, est inappropriée. Dans notre travail (Chapitre 4), nous proposons une approche basée sur les *automates de mots*, qui permet l'évaluation des requêtes positives de Core XPath, sur les documents XML compressés, représentés par des dags (cette approche a été publiée dans [30]). Nous montrons (Sections 4.5 et 4.6), que la complexité en temps d'évaluation d'une requête Q sur un document (compressé ou non) t, en utilisant cette approche, est linéaire par rapport à la taille de Q et au nombre d'arêtes de t.

Il est évident que le dag minimal correspondant à un arbre donné t, peut être produit en temps linéaire par rapport à la taille de t. Une telle représentation entraîne, dans le meilleur des cas, un gain exponentiel d'espace de stockage — il suffit de considérer un arbre binaire où tous les nœuds différents de la racine portent le même nom. Néanmoins, d'autres techniques de compression, plus efficaces, sont connues. C'est le cas de la compression basée sur une grammaire hors contexte d'arbre ([25]), appelée *straightline* (SL), qui a été proposée dans le contexte de XML par Busatto et Maneth dans [61]. La notion d'une grammaire SL a été auparavant utilisée dans le cas des mots [80, 83, 56]. L'idée, dans le cas des grammaires d'arbres, est de partager non seulement des sous–arbres communs, mais aussi des *parties internes* d'un arbre considéré. Les auteurs de [18] présentent un algorithme dit BPLEX, qui produit, en temps linéaire par rapport à la taille d'un arbre t, une grammaire SL hors contexte, représentant t. Théoriquement, une telle représentation peut mener jusqu'à un gain doublement exponentiel d'espace de stockage, ce qui est confirmé par des résultats expérimentaux (voir [18]).

Frick, Grohe et Koch prouvent, dans [35], que le problème d'évaluation des requêtes de Core XPath sur les documents XML compressés, représentés à l'aide des dags, est PSPACE–complete. Les auteurs de [59] étendent ce résultat à la compression basée sur des grammaires SL hors contexte d'arbres.

Les résultats sur la compression présentés plus haut ne tiennent compte que de la structure arborescente — le squelette du document XML. Une question naturelle se pose : que faire avec le PCDATA, l'information textuelle stockée aux feuilles de tels documents? Deux différentes approches ont été développées pour traiter ce

problème. La première consiste à détacher la structure du document (arbre) de son contenu (PCDATA), et compresser chacune de ces parties séparément. Cette approche (appelée *permutation–based*) permet d'obtenir de très bons résultats au niveau de compression, et elle est utilisée dans la plupart d'outils de compression de XML : XMill [55], XComp [54], XWRT [86], AXECHOP [52]. La deuxième approche, appelée *homomorphique*, consiste à compresser chaque XML token individuellement, ce qui permet de maintenir la structure du document d'origine. Les outils qui ont été implémentés en utilisant ce principe sont, entre autres : XMLPP [23], DTDPPM [24], et SCMPPM [3].

En général, les méthodes de compression des documents XML peuvent être classées en deux groupes :

- *non–queryable* — inappropriées à l'évaluation des requêtes,
- *queryable* — appropriées à l'évaluation des requêtes.

Les approches appartenant au premier groupe servent à produire des représentations très économes par rapport à l'espace de stockage. Ces représentations favorisent l'archivage des données massives qui peuvent être plus rapidement échangées (p.ex., sur le net). Tous les outils mentionnés dans le paragraphe précédent sont basés sur ce type de compression. Les approches appartenant au deuxième groupe fournissent des formes compressées de plus grande taille, mais sur lesquelles des requêtes peuvent être évaluées efficacement sans la nécessité d'une décompression préalable. Parmi les outils de compression fournissant des représentations appropriées à l'évaluation des requêtes, on peut citer les suivants : XQueC [8], BPLEX [18], TREECHOP [53], XGRIND[88], XPRESS [69]

Un compendium exhaustif sur les différentes techniques de compression des données massives peut être trouvé sur le site Internet de Dagstuhl Seminar 08261 *Structure–Based Compression of Complex Massive Data*, à l'adresse ci–dessous : `http://kathrin.dagstuhl.de/08261/Materials2/`.

1.4. Inclusion et minimisation de requêtes

Les questions d'inclusion et d'équivalence des schémas de requêtes (pattern containment and equivalence) [6 , , 89, 85] sont étroitement liées à celle d'évaluation efficace. Elles peuvent permettre de résoudre un problème très important, celui de la minimisation de requêtes [33]. Puisque le temps nécessaire pour évaluer une requête donnée Q dépend de la taille de Q ([41]), la minimisation — possibilité de remplacer Q par une expression équivalente, mais ayant la plus petite taille possible — est cruciale, et a toujours attiré l'attention de nombreux chercheurs : d'abord pour des requêtes relationnelles [21], et dernièrement pour des requêtes sur les documents XML [33, 62, 4, 5, 90, 48, 82].

Le fragment de XPath considéré le plus souvent dans les travaux mentionnés est noté XP(`/`,`//`,`[ ]`,$*$), et est composé des expressions de Core XPath n'utilisant que les axes descendants `child` (`/`) et `descendant` (`//`), les filtres qualificatifs (`[ ]`) et le symbole '$*$' (wildcard) de XPath. Bien que XP(`/`,`//`,`[ ]`,$*$) ne couvre pas le XPath (ni Core XPath) tout entier, il constitue un fragment important du point de vue des applications — il contient suffisamment d'opérateurs pour exprimer d'une

façon naturelle des chemins d'accès. Par conséquent, c'est un fragment le plus souvent utilisé de tout XPath. Toute expression de XP(/,//,[ ],$*$) est une requête qui peut être représentée par un graphe arborescent, appelé *pattern unaire* ou tout simplement *pattern*, ayant deux types d'arêtes (simples pour `child` et doubles pour `descendant`), dont les nœuds sont étiquetés par des symboles d'un alphabet donnée Σ ou par '$*$', et où on a un nœud distingué (output node d'une requête *unaire*), représentant l'information sélectionnée par l'expression en question. D'une manière générale (voir Chapitre 5 pour la définition formelle), on dit qu'un pattern P est inclus dans un pattern Q ($P \subseteq Q$) ssi l'information exprimée par Q est plus générale que celle exprimée par P. Deux patterns sont équivalents ($P \equiv Q$) ssi les requêtes correspondantes sont équivalentes, c.–à–d., fournissent exactement la même réponse sur tout document. Autrement dit, l'équivalence de deux patterns $P \equiv Q$ n'est rien d'autre que la conjonction de deux inclusions $P \subseteq Q$ et $Q \subseteq P$. Mikalu et Suciu montrent dans [6] que le problème d'inclusion de deux patterns est réductible en temps polynomial à celui d'équivalence de deux patterns. Par suite, tout les résultats sur l'inclusion s'appliquent aussi au problème d'équivalence.

Les auteurs de [6] montrent que le problème d'inclusion de patterns du fragment XP(/,//,[ ],$*$) est coNP–complet. Ils fournissent un algorithme correct et complet, qui en temps exponentiel permet de vérifier si pour deux patterns P et Q du fragment mentionné on a $Q \subseteq P$. Ils introduisent également la notion d'homomorphisme entre deux patterns (voir Chapitre 5 pour la définition), et prouvent que s'il existe un homomorphisme de P vers Q, alors $Q \subseteq P$. Ils montrent que c'est une condition suffisante mais pas nécessaire pour l'inclusion sur le fragment XP(/,//,[ ],$*$). Ils présentent un algorithme, polynomial par rapport à la taille des patterns P et Q, pour vérifier l'existence d'un homomorphisme de P vers Q. Par conséquent, ils obtiennent un algorithme pour vérifier l'inclusion $Q \subseteq P$, qui est polynomial, correct mais incomplet. Ils examinent également des cas spéciaux, où $Q \in$ XP(/,[ ],$*$), $P \in$ XP(/,[ ],$*$), $P \in$ XP(/,//,[ ]), ou encore $P \in$ XP(/,//,$*$), et justifient que dans toutes ces situations leurs algorithme est complet.

Ce qui est à l'origine de la complétude du problème d'inclusion sur le fragment XP(/,//,[ ],$*$) tout entier, c'est la possibilité d'utiliser en même temps l'axe `descendant` et le symbole don't–care '$*$'. Les résultats obtenus dans les travaux antérieurs [4, 68, 100] prouvent que pour tout strict sous–fragment de XP(/,//,[ ],$*$), obtenu en interdisant l'utilisation d'un des opérateurs //,[ ] ou $*$, il existe des algorithmes efficaces pour vérifier l'inclusion. Flesca et al. considèrent dans [33] le fragment de XP(/,//,[ ],$*$), composé seulement de patterns où chaque nœud portant le nom '$*$' a au plus un enfant. Ce fragment est noté XP([$\not=$]). Les auteurs de [33] montrent que le problème d'inclusion sur XP([$\not=$]) reste coNP–complet. Le tableau ci–dessous, basé sur [33], donne une synopsis sur la complexité du problème d'inclusion de patterns sur les différents fragments de XPath :

Fragment	Complexité	Référence
XP(/,//,[ ],∗)	coNP–complet	[6]
XP(/,//,∗)	P	[68]
XP(/,[ ],∗)	P	[6 , 100]
XP(/,//,[ ])	P	[5]
XP([∗])	coNP–complet	[33]

Une synthèse plus détaillée sur les algorithmes d'inclusion et leurs complexités se trouve dans [85] et [33].

Il faut évoquer encore un aspect intéressant qui résulte du concept de voir une requête comme un pattern : une telle vision permet, d'une façon naturelle, de représenter des requêtes n–aires, c.–à–d., des requêtes auxquelles la réponse est composée des n–uplètes de nœuds [8, 48]. Dans ce cas on parle d'un *pattern n–aire*, où (au lieu d'un seul) on a n nœuds distingués représentant l'information sélectionnée par la requête correspondante [6 , 48]. La représentation des requêtes sous forme des patterns offre ainsi une possibilité de définir un cadre uniforme pour le traitement de requêtes d'arité quelconque. L'utilisation des patterns n–aires n'est pas plus difficile ni plus complexe que celle des patterns unaires. En effet, Miklau et Suciu montrent dans [6], que tout pattern n–aire sur un alphabet donnée Σ peut être transformé en un pattern *booléen* (n'ayant aucun nœud distingué) sur un alphabet Σ' approprié. Ils prouvent également, que pour le besoin du problème d'inclusion, il suffit de ne considérer que des patterns booléens. Il semble qu'un résultat similaire peut être obtenu pour résoudre d'autres problèmes, entre autres, l'évaluation de requêtes.

Dans le Chapitre 5, nous présentons une méthode basée sur des techniques de réécriture, pour résoudre le problème d'inclusion sur XP(/,//,[ ],∗). Nous définissons, en termes des règles de réécriture, une condition *nécessaire et suffisante* pour vérifier l'inclusion de deux patterns du fragment mentionné. Cette approche, qui a été également publié dans [50], peut être facilement adapté au problème d'évaluation de requêtes d'arité quelconque.

Chapitre 2

Requêtes sur documents XML — préliminaires

Dans ce chapitre nous présentons le cadre de notre travail : les documents XML et le langage XPath. Nous supposons que le lecteur est familiarisé avec ces deux formalismes, dont nous ne présentons ici que des concepts généraux. Après une courte introduction aux documents XML (Section 2.1), nous introduisons quelques notions utilisées dans la suite de ce rapport (Section 2.2). La Section 2.3 est consacrée à une présentation générale du langage XPath. Dans les Sections 2.4 et 2.5 nous définissons, en détail, deux fragments de XPath, auxquels nous nous intéresserons plus particulièrement dans notre travail.

2.1. Documents XML

XML — Langage de Balisage Extensible (en anglais *eXtensible Markup Language*) — a été développé par un groupe de travail XML, présidé par Jon Bosak, en 1996. Comme le mentionne [9], les objectifs de conception de XML étaient les suivants :

- XML devrait pouvoir être utilisé sans difficulté sur Internet,
- XML devrait soutenir une grande variété d'applications,
- XML devrait être compatible avec SGML [91],
- il devrait être facile d'écrire des programmes traitant les documents XML,
- le nombre d'options dans XML doit être réduit au minimum, idéalement à aucune,
- les documents XML devraient être lisibles par l'homme, et raisonnablement clairs,
- la conception de XML devrait être préparée rapidement,
- la conception de XML sera formelle et concise,
- il devrait être facile de créer des documents XML.

Le langage XML décrit une classe d'objets de données, appelés *documents XML*. La structure de stockage d'un document XML (sa structure logique) est décrite par des *balises*. Un document XML est composé d'un ou de plusieurs *éléments*, dont les limites sont marquées par des *balises ouvrantes* et *fermantes*. Chaque élément a un type identifié par un nom. Les éléments peuvent être imbriqués, c.–à–d., un élément peut contenir d'autres éléments. On utilise des *attributs*, pour associer des couples (attribut,valeur) aux éléments. Les documents XML peuvent être modélisés à l'aide des arbres ordonnés et étiquetés, dont les étiquettes sont d'arité non fixe, et les noeuds représentent les éléments. La Figure 2.1 illustre un simple document XML représentant un réseau d'espionnage.

Un document XML peut être valide par rapport à une DTD (Document Type Definition), ou un schéma (XML Schema) donné. Une DTD est un document SGML, qui indique, entre autres, quel doit être le contenu de chaque élément d'un document

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<espions>
    <personne nationalité="PL">
        <nom> Kukliński Ryszard </nom>
        <époque> 1981 </époque>
    </personne>
    <personne nationalité="USA">
        <nom> Ames Alrdich </nom>
        <époque> 1983 </époque>
    </personne>
    <personne nationalité="FR">
        <surnom> Chevalier d'Éon </surnom>
    </personne>
</espions>
```

Figure 2.1. Document XML

XML. Une DTD spécifie le contenu d'un élément, en indiquant son nom, ses attributs, ainsi que l'ordre et le nombre d'occurrences autorisées des sous–éléments. L'ensemble constitue la définition des hiérarchies valides d'éléments et de texte. Voici ce que pourrait–être la DTD du document représenté sur la Figure 2.1 :

```
<!ELEMENT espions (personne*)>
<!ELEMENT personne (nom | surnom, époque?)>
<!ATTLIST personne nationalité (PL | DE | USA | FR | RUS) "RUS">
<!ELEMENT nom (#PCDATA)>
<!ELEMENT surnom (#PCDATA)>
<!ELEMENT époque (#PCDATA)>.
```

XML Schema [95] est un langage de description du format de document XML, permettant de définir la structure d'un document XML. Une instance d'un XML Schema est elle–même un document XML, qui étend les possibilités offertes par les DTD. Il permet par exemple, de définir des domaines de validité pour la valeur d'un champ, ce qui n'est pas possible dans une DTD.

2.2. Représentation de documents

Dans cette section on établit quelques notions générales, qui seront utilisées dans la suite de ce document pour modéliser les documents XML.

Considérons un graphe orienté $G = (V, E)$, où V est l'ensemble des sommets, et $E \subseteq V \times V$ l'ensemble des arêtes. Soient deux sommets $u, v \in V$, on dit que :

- u est *enfant* de v, si l'arête (v, u) appartient à l'ensemble E,
- v est *père* de u, si u est un enfant de v,
- v est *racine* de G, si v ne possède aucun père,
- v est *feuille* de G, si v ne possède aucun enfant.

Pour chaque sommet v de G, on pose $Parents(v) = \{w \in V \mid w \text{ est père de } v\}$. On définit, de façon récursive, les notions des sommets descendants et ancêtres :

- u est *descendant* de v, si u est un enfant de v, ou il existe un sommet $w \in V$ enfant de v, tel que u soit un descendant de w;
- u est *ancêtre* de v, si u est un père de v, ou il existe un sommet $w \in V$ père de v, tel que u soit un ancêtre de w.

Par $|G|$ on désignera la *taille* du graphe G, c.–à–d., le *nombre d'arêtes* de G. Tous les graphes considérés dans ce travail seront orientés et sans cycles. Il est alors plus légitime d'appeler les éléments de V des *nœuds* au lieu de sommets.

Par Σ on désignera un ensemble de symboles, appelé *alphabet*, composé de noms d'éléments de tous les documents XML considérés. Bien qu'on suppose que Σ est un ensemble fini, tous les résultats décrits dans ce travail sont aussi valables dans le cas d'un alphabet infini. Puisqu'il s'agit de concevoir des approches pour le traitement des documents XML, on supposera que les symboles de Σ n'ont pas d'arité fixe ni bornée.

Rappelons, qu'un *arbre* est un graphe $t = (Nodes_t, Edges_t)$ orienté et sans cycles, tel que :

- t ait une seule racine, notée $root_t$,
- tout nœud $v \in Nodes_t$, différent de la racine, possède un seul père.

Par définition, tout document XML à une structure arborescente, et d'une manière naturelle, peut être représenté par un *arbre étiqueté* $t = ((Nodes_t, Edges_t), name_t)$, où $name_t$: $Nodes_t \to \Sigma$ est une fonction assignant à chaque nœud v de t un nom $name_t(v) \in \Sigma$, correspondant au nom d'élément représenté par v. Par abus de langage, on confondra souvent le document XML t avec l'arbre représentant sa structure, et on l'appellera tout simplement *arbre XML* t. Tout arbre XML t contiendra une racine supplémentaire $Root_t$, appelé *racine fictive*, et représentant ce qu'on appelle *élément document* (en anglais *document element*).

Dans la Section 4.1 on montrera comment modéliser les documents XML compressés, en utilisant la structure de dags (directed acyclic graphs) étiquetés. L'*ordre du document* (en anglais *document order*) impose que tout graphe représentant un document XML doit être *ordonné*, c.–à–d., que des arêtes sortantes de chaque nœud doivent être ordonnées. Cet ordre permet de définir la notion d'un nœud frère sur un document XML. Soit t un graphe ordonné (arbre ou dag) représentant un document XML. Pour tout nœud v de t, notons par $\gamma(v) = (u_1, u_2, \ldots, u_n)$ une suite composée de tous les enfants de v. Considérons un indice $i \in \{1, \ldots, n\}$, et un nœud $u_i \in \gamma(v)$:

- tout nœud $u_j \in \gamma(v)$, tel que $i < j$ (resp. $j < i$) est appelé *frère suivant* (resp. *frère précédent*) de u_i;
- si $j = i + 1$ (resp. $j = i - 1$), on dit que u_j est *frère suivant immédiat* (resp. *frère précédent immédiat*) de u_i.

2.3. Requêtes

L'objectif du travail présenté dans ce document, nous l'avons dit, est l'évaluation de requêtes sur les documents XML classiques (arborescents), compressés (représentés à l'aide de DAGs), ou encore assujettis à des politiques du contrôle d'accès.

Dans le contexte des bases de données (notamment des bases de données XML), le terme *requête* (en anglais *query*) correspond à une interrogation d'une base ou d'un document, pour en récupérer une certaine partie des données. Par *évaluation d'une requête* Q sur un document XML t, on comprendra la sélection de tous les nœuds de t, qui satisfont les propriétés exprimées par Q. Soit une requête Q, et un document t. Tout nœud de t, satisfaisant les propriétés exprimées par Q, sera appelé une *réponse* à Q sur t, ou encore *nœud sélectionné par* Q. L'ensemble de tous les nœuds de t sélectionnés par Q, sera noté R^t_Q, et appelé *REPONSE* à Q sur t. Deux requêtes Q et Q' données sont dites *équivalentes*, ssi pour tout document t, on a $R^t_Q = R^t_{Q'}$.

Il existe plusieurs formalismes pour exprimer les requêtes (voir Section 1.1). Dans ce travail, on utilise le langage XPath, notamment XPath Version 1.0, introduit par The World Wide Web Consortium (W3C) dans [92]. On suppose que le lecteur a une connaissance de base de la syntaxe de XPath, cf. [92, 16]. XPath est un langage conçu pour adresser et sélectionner des parties de documents XML. Il est basé sur des axes dits *de navigation*, parmi lesquels nous retenons : `self`, `child`, `parent`, `ancestor`, `descendant`, `following-sibling`, et `preceding-sibling`. Les sept axes énumérés ci–dessus seront appelés *axes de base* : les cinq premiers servent à naviguer suivant les chemins racine–feuille d'un document donné, et seront nommés *axes verticaux*; les deux derniers permettent une navigation entre les frères, et seront nommés *axes horizontaux*. Les autres axes de navigation de XPath sont : `ancestor-or-self`, `descendant-or-self`, `following` et `preceding`. On peut les exprimer à l'aide de connecteurs logiques et les sept axes de base — pour tout σ d'un alphabet donné Σ, on a :

- `/ancestor-or-self::`$\sigma \equiv$`/ancestor::`σ *or* `/self::`σ,
- `/descendant-or-self::`$\sigma \equiv$`/descendant::`σ *or* `/self::`σ,
- `/following::`$\sigma \equiv$`/ancestor-or-self::`$*$`/following-sibling::`$*$
  `/descendant-or-self::`σ,
- `/preceding::`$\sigma \equiv$`/ancestor-or-self::`$*$`/preceding-sibling::`$*$
  `/descendant-or-self::`σ.

Parfois, on utilisera aussi les axes `right` et `left`, qui correspondent respectivement au frère suivant immédiat et frère précédent immédiat. En XPath, on les définit de la façon suivante :

- `/right::`$\sigma \equiv$`/following-sibling::`σ`[`$position() = 1$`]`,
- `/left::`$\sigma \equiv$`/preceding-sibling::`σ`[`$position() = 1$`]`.

Considérons un document XML t. Tout axe de navigation de XPath peut être vu comme une relation binaire sur $Nodes_t$. Soient deux nœuds u, v de t, et un axe de navigation `axis`. On dit que

$$v \,\texttt{axis}\, u \quad \textit{est satisfait sur } t$$

si et seulement si, en partant du nœud v, et en suivant l'axe `axis`, on peut arriver au nœud u. Par exemple, $v \,\texttt{child}\, u$ est satisfait sur t si et seulement si, u est un enfant de v sur t. Dans le suite, on notera souvent $v \,\texttt{axis}\, u$, pour dire que $v \,\texttt{axis}\, u$ est satisfait sur t. On définit *axe inverse* d'axe `axis`, comme un unique axe, noté $\texttt{axis}^{-1}$, qui vérifie la condition suivante :

pour tout $u, v \in Nodes_t$: $v\,\texttt{axis}^{-1}\,u$ si et seulement si $u\,\texttt{axis}\,v$.

Par *direction* d'axe `axis`, on comprendra un axe noté `dir-axis`, qui définit le sens d'action d'axe `axis`. La Table 2.1 présente les axes inverses ainsi que les directions des sept axes de base de XPath.

axe	axe inverse	direction
`axis`	$\texttt{axis}^{-1}$	`dir-axis`
`self`	`self`	`self`
`parent`	`child`	`parent`
`child`	`parent`	`child`
`descendant`	`ancestor`	`child`
`ancestor`	`descendant`	`parent`
`following-sibling`	`preceding-sibling`	`right`
`preceding-sibling`	`following-sibling`	`left`

Table 2.1. Axes de base, leurs axes inverses et directions

Toute *requête* Q de XPath est une formule basée sur un chemin d'accès. L'ensemble composé de tous les nœuds d'un document donné t, qui vérifient ce chemin d'accès, forme la REPONSE à Q sur t. Dans les deux sections suivantes, on présente des fragments de langage XPath auxquels on s'intéresse dans ce travail.

2.4. Requêtes positives de Core XPath

Core XPath est un fragment de navigation de langage XPath. Les expressions de Core XPath ne tiennent pas compte des données aux nœuds de documents. On considérera ici seulement les formules *positives*, c.–à–d., sans symbole de négation. Ce fragment est suffisant pour couvrir les requêtes qui ne regardent que la structure des documents XML. Il sera utilisé dans le Chapitre 4, où on présente une approche permettant d'évaluer certaines requêtes positives de Core XPath sur les documents pouvant être donnés sous une forme compressée.

Soit un alphabet donné Σ, contenant les noms d'éléments de tous les documents considérés. Les requêtes qui nous intéressent, sont des expressions Q_{can} générées à partir de la grammaire présentée dans la Table 2.2, où `A` est un des sept axes de base de XPath, $x \in \Sigma \cup \{*\}$, et le symbole '$*$' est le *wildcard* de XPath, pouvant remplacer tout élément de Σ. Les requêtes Q_{can}, obtenues à partir de cette grammaire, seront

$$\begin{array}{lcl} L_{can} & : & position() = i \mid true \\ S_{can} & : & true \mid \texttt{A::}x\texttt{[}L_{can}\texttt{]} \mid S_{can}\ and\ S_{can} \mid S_{can}\ or\ S_{can} \\ E_{can} & : & \texttt{A::}x\texttt{[}L_{can}\texttt{][}S_{can}\texttt{]} \mid \texttt{A::}x\texttt{[}L_{can}\texttt{][}E_{can}\texttt{]} \\ Q_{can} & : & /E_{can} \mid Q_{can}Q_{can} \end{array}$$

Table 2.2. Grammaire pour des requêtes canoniques

appelées *requêtes en forme canonique*, ou *requêtes canoniques*. On nomme *filtre qualificatif* (ou *filtre*), toute expression de la forme $\texttt{[}X_{can}\texttt{]}$, où $X_{can} \in \{L_{can}, S_{can}, E_{can}\}$. On suppose que le filtre $\texttt{[}true\texttt{]}$ est vrai à chaque nœud de tous les documents considérés; on identifie l'expression $\texttt{A::}x\texttt{[}true\texttt{]}$ avec $\texttt{A::}x$. Soit un document t. La requête canonique la plus simple est de la forme $Q_{can} = \texttt{/A::}x$. Pour l'évaluer sur un document t, on se place à la racine fictive de t (symbole '/'), et on cherche tous les nœuds de t, qui vérifient le chemin d'accès défini par ce qu'on appelle *l'étape de localisation* (en anglais *location step*) $\texttt{A::}x$. D'où, un nœud v de t constitue une réponse à $Q_{can} = \texttt{/A::}x$, si et seulement si :

- $Root_t$ $\texttt{A}$ v est satisfait sur t,
- et le nom de v est x.

Par exemple, la REPONSE à la requête $\texttt{/descendant::}a$ sur un document donné t, est composée de tous les nœuds de t (descendants de la racine fictive), dont le nom est a. Notons que si $x = *$, alors tous les nœuds v satisfaisant $root_t$ $\texttt{A}$ v sur t constituent des réponses à Q. Bien évidement, la requête peut être plus complexe. C'est le cas des requêtes *imbriquées*, c.–à–d., de la forme $\texttt{/A::}x\texttt{[}L_{can}\texttt{]}\texttt{[}S_{can}\texttt{[...[}S_{can}\texttt{]...]]}$. Par exemple, la REPONSE à la requête $\texttt{/descendant::}a\texttt{[child::}b\texttt{]}$ sur un document t, est composée de tous les nœuds de t, ayant le nom a et vérifiant le filtre $\texttt{[child::}b\texttt{]}$ (c.–à–d., ayant un enfant portant le nom b). Les expressions de la forme $\texttt{/}E_{can}$ seront appelées *requêtes élémentaires*. La dernière production de la grammaire présentée dans la Table 2.2, implique que toute requête canonique Q_{can} est une concaténation d'un certain nombre de requêtes élémentaires. Il n'est pas difficile de remarquer que la forme générale d'une requête canonique Q_{can} est :

$$Q_{can} = /C_1/\ldots/C_n,$$

pour un certain $n \in \mathbb{N}$, où chaque requête élémentaire $/C_i$ est de la forme

$$\texttt{/A::}x\texttt{[}L_{can}\texttt{]}\texttt{[}X_{can}\texttt{]},$$

où $X_{can} \in \{S_{can}, E_{can}\}$. Le nombre de requêtes élémentaires composant la requête Q est appelé *profondeur* de Q. Il est important de ne pas confondre la profondeur d'une requête avec sa taille. Par *taille* d'une requête Q, notée $|Q|$, on comprend le nombre d'étapes de localisation $\texttt{A::}x\texttt{[}L_{can}\texttt{]}$ figurant dans Q.

Exemple 2.1. *La requête* $Q = \texttt{/descendant::}a\texttt{/child::}b\texttt{[ancestor::}c\texttt{]}$ *est de profondeur* 2*, mais sa taille est* 3*. Pour trouver la REPONSE à cette requête sur un document* t*, on procède comme suit :*

- *on se place à la racine fictive de* t*,*
- *en descendant vers tous les nœuds de* t*, on choisit ceux qui portent le nom* a*,*
- *ensuite, on sélectionne les enfants de ces nœuds, qui portent le nom* b*,*
- *finalement, parmi ces nœuds on garde dans la REPONSE* R^t_Q *seulement ceux qui ont au moins un ancêtre* c*.*

Notons encore, que le prédicat $\texttt{[}position() = i\texttt{]}$ permet de sélectionner des nœuds qui se trouvent à la i–ème position par rapport à l'axe considéré : ainsi la requête $\texttt{/descendant::}a\texttt{/child::}b\texttt{[}position() = 2\texttt{]}$ sélectionne tous les nœuds b d'un document donné, ayant un père a et exactement un seul frère précédant b.

On introduit maintenant une autre forme pour les requêtes positives de Core XPath, appelée *forme standardisée*. Nous utiliserons les requêtes de cette forme au Chapitre 4, pour interroger les documents XML compressés. La Table 2.3 présente la grammaire pour générer des requêtes en forme standardisée. Le symbole '//' est

$$
\begin{array}{lcl}
L_{std} & : & position() = i \mid true \\
S_{std} & : & Root \mid \texttt{A::}x \mid \texttt{A::}x\texttt{[}L_{std}\texttt{]} \mid S_{std}\ and\ S_{std} \mid S_{std}\ or\ S_{std} \\
E_{std} & : & \texttt{A::*[}S_{std}\texttt{]} \mid \texttt{A::*[}L_{std}\texttt{][}S_{std}\texttt{]} \mid \texttt{A::*[}E_{std}\texttt{]} \mid \texttt{A::*[}L_{std}\texttt{][}E_{std}\texttt{]} \\
Y_{std} & : & S_{std} \mid E_{std} \mid Y_{std}\ and\ Y_{std} \mid Y_{std}\ or\ Y_{std} \\
Q_{std} & : & \texttt{//*} \mid \texttt{//*[}Y_{std}\texttt{]}
\end{array}
$$

Table 2.3. Grammaire pour des requêtes standardisées

utilisé ici à la place de l'axe `descendant`. Toute requête Q_{std}, générée par cette grammaire, sera appelée *requête en forme standardisée*, ou *requête standardisée*. La forme générale d'une requête standardisée est $\texttt{//*[}Y_{std}\texttt{]}$, où Y_{std} est soit S_{std} ou E_{std}, soit une conjonction ou une disjonction des expressions de ce type. La sémantique de L_{std}, S_{std}, E_{std} et Q_{std} est évidente, et correspond respectivement à celle de L_{can}, S_{can}, E_{can} et Q_{can}. La REPONSE à la requête standardisée $Q_{std} = \texttt{//*[}Y_{std}\texttt{]}$, sur un document t, est composée des nœuds de t qui vérifient le filtre $\texttt{[}Y_{std}\texttt{]}$ (qui n'est autre qu'un prédicat) . Par une simple observation des deux grammaires ci–dessus, on obtient la remarque suivante :

Remarque 2.1. *1. La profondeur de toute requête standardisée est égale à* 1.
2. Toute requête standardisée est aussi canonique.

La réciproque du point 2 de la Remarque 2.1, est fausse (considérer par exemple, $Q = \texttt{/child::}a$). Pourtant, pour chaque requête canonique, il existe une requête standardisée équivalente. Pour pouvoir exprimer la requête équivalente à $\texttt{/child::}a$, on utilise le test *Root* (voir la grammaire, Table 2.3), qui est vrai à la racine fictive $Root_t$ de tout document t. Ainsi, la requête $\texttt{//*[parent::}Root\ and\ \texttt{self::}a\texttt{]}$ est en forme standardisée, équivalente à la requête canonique $\texttt{/child::}a$.

On présente maintenant une procédure, qui à partir d'une requête en forme canonique Q_{can}, produit (en temps linéaire par rapport à la taille de Q_{can}) une requête équivalente en forme standardisée, notée $Std(Q_{can})$. Cette procédure est récursive. On montre d'abord comment calculer la forme standardisée d'une expression $\texttt{[}X_{can}\texttt{]}$, notée $Std(\texttt{[}X_{can}\texttt{]})$, où $X_{can} \in \{L_{can}, S_{can}, E_{can}\}$:

$$
\begin{array}{c}
Std(\texttt{[}L_{can}\texttt{]}) = L_{can}; \\
Std(\texttt{[}S_{can}\texttt{]}) = S_{can};
\end{array}
$$

pour calculer la fome standardisée de $\texttt{[}E_{can}\texttt{]}$, on procède récursivement comme suit :

$$
\begin{array}{l}
Std(\texttt{[A::}x\texttt{[}L_{can}\texttt{][}S_{can}\texttt{]]}) = \texttt{A::*[self::}x\ and\ Std(\texttt{[}L_{can}\texttt{])}\ and\ Std(\texttt{[}S_{can}\texttt{])]}; \\
Std(\texttt{[A::}x\texttt{[}L_{can}\texttt{][}E_{can}\texttt{]]}) = \texttt{A::*[self::}x\ and\ Std(\texttt{[}L_{can}\texttt{])}\ and\ Std(\texttt{[}E_{can}\texttt{])]}.
\end{array}
$$

Notons que $\texttt{axis::}x\ and\ \texttt{[}L_{can}\texttt{]}$ est ici une notation alternative pour l'expression $\texttt{axis::}x\texttt{[}L_{can}\texttt{]}$. Par conséquent, la forme standardisée de l'expression imbriquée

$$\texttt{[A::}\sigma\texttt{[}L^0_{can}\texttt{][A}_1\texttt{::}\sigma_1\texttt{[}L^1_{can}\texttt{][...[A}_k\texttt{::}\sigma_k\texttt{[}L^k_{can}\texttt{]]...]]]}$$

est la suivante :

$$Std(\texttt{[A::}\sigma\texttt{[}L^0_{can}\texttt{]][A}_1\texttt{::}\sigma_1\texttt{[}L^1_{can}\texttt{]][...[A}_k\texttt{::}\sigma_k\texttt{[}L^k_{can}\texttt{]]...]]]}) =$$
$$\texttt{A::*[self::}\sigma \textit{ and } Std(\texttt{[}L^0_{can}\texttt{]}) \textit{ and } \texttt{A}_1\texttt{::*[self::}\sigma_1 \textit{ and } Std(\texttt{[}L^1_{can}\texttt{]}) \textit{ and...}$$
$$\texttt{A}_{k-1}\texttt{::*[self::}\sigma_{k-1} \textit{ and } Std(\texttt{[}L^{k-1}_{can}\texttt{]}) \textit{ and } \texttt{A}_k\texttt{::}\sigma_k\texttt{[}L^k_{can}\texttt{]]...]]}.$$

Pour une requête en forme standardisée $Q = \texttt{//*[}X\texttt{]}$, on note par $exp(Q)$ l'expression X. On utilise *conn* pour dénoter les connecteurs logiques *and*, *or*. Considérons une requête canonique $Q_{can} = /C_1/C_2/\ldots/C_n$, de profondeur n. Voici comment générer une requête standardisée $Std(Q_{can})$, équivalente à Q_{can} :

Procédure de conversion de Q_{can} vers $Std(Q_{can})$

Cas de profondeur $n = 1$: $Q_{can} = /C_1$
$Std(\texttt{/child::}\sigma\texttt{[}X_{can}\texttt{]}) = \texttt{//*[(parent::}Root \textit{ and } \texttt{self::}\sigma\texttt{)} \textit{ and } Std(\texttt{[}X_{can}\texttt{]})\texttt{]}$;
$Std(\texttt{/child::*[}X_{can}\texttt{]}) = \texttt{//*[parent::}Root \textit{ and } Std(\texttt{[}X_{can}\texttt{]})\texttt{]}$;
$Std(\texttt{/descendant::}\sigma\texttt{[}X_{can}\texttt{]}) = \texttt{//*[self::}\sigma \textit{ and } Std(\texttt{[}X_{can}\texttt{]})\texttt{]}$;
$Std(\texttt{/descendant::*[}X_{can}\texttt{]}) = \texttt{//*[}Std(\texttt{[}X_{can}\texttt{]})\texttt{]}$.

Remarquons que dans le cas d'une requête de profondeur 1, il suffit de considérer les axes `child` et `descendant`, car pour tout autre axe de base `axis`, la requête $\texttt{axis::}x\texttt{[}X_{can}\texttt{]}$ admet la REPONSE vide sur tout document. Pour traiter le cas des requêtes de profondeur $n > 1$, on utilise la notion d'axes inverses (voir la Table 2.1) :

Cas de profondeur $n > 1$: $Q_{can} = /C_1/C_2/\ldots/C_n$
$Std(/C_1/\ldots/C_{n-1}/\texttt{A::}\sigma\texttt{[}X_{can}\texttt{]}) =$
$\texttt{//*[(self::}\sigma \textit{ and } Std(\texttt{[}X_{can}\texttt{]})) \textit{ and } \texttt{A}^{-1}\texttt{::*[}exp(Std(/C_1/\ldots/C_{n-1}))\texttt{]]}$.

La taille de la requête $Std(Q_{can})$, équivalente à Q_{can}, produite en utilisant la procédure ci–dessus, est $|Std(Q_{can})| = 2*|Q_{can}| \in \mathcal{O}(Q_{can})$. La sémantique d'axes inverses implique que, pour tout document t, on a $R^t_{Q_{can}} = R^t_{Std(Q_{can})}$.

Remarquons que les requêtes canoniques considérées dans ce travail (Table 2.2) ne permettent l'utilisation des connecteurs logiques $conn \in \{and, or\}$ que dans les filtres. Dans le cas où on autoriserait les requêtes avec *conn* dans la partie de navigation, la transformation d'une requête canonique en une requête standardisée équivalente aurait un coût exponentiel, ce que montre le raisonnement suivant :

$Std(\texttt{/axis::}\sigma\texttt{[}X_{can}\texttt{]} \textit{ conn } \texttt{axis'::}\sigma'\texttt{[}X'_{can}\texttt{]} =$
$\texttt{//*[}exp(Std(\texttt{/axis::}\sigma\texttt{[}X_{can}\texttt{]})) \textit{ conn } exp(Std(\texttt{/axis'::}\sigma'\texttt{[}X'_{can}\texttt{]}))\texttt{]}$;
$Std(/C_1/\ldots/C_{n-1}/\texttt{A::}\sigma\texttt{[}X_{can}\texttt{]} \textit{ conn } \texttt{A'::}\sigma'\texttt{[}X'_{can}\texttt{]}) =$
$\texttt{//*[((self::}\sigma \textit{ and } Std(\texttt{[}X_{can}\texttt{]})) \textit{ and } \texttt{A}^{-1}\texttt{::*[}exp(Std(/C_1/\ldots/C_{n-1}))\texttt{])} \textit{ conn}$
$\texttt{((self::}\sigma' \textit{ and } Std(\texttt{[}X'_{can}\texttt{]})) \textit{ and } \texttt{A'}^{-1}\texttt{::*[}exp(Std(/C_1/\ldots/C_{n-1}))\texttt{])]}$.

2.5. Requêtes positives de XPath

Les requêtes qu'on peut générer à partir des grammaires des Tables 2.2 et 2.3, ne regardent que la structure des documents. Dans cette section, on présente un fragment plus large de XPath, dont les requêtes prennent en compte aussi des données textuelles, ainsi que des valeurs d'attributs stockées aux nœuds de documents. On utilisera ce fragment de XPath dans le Chapitre 3, où on présente une méthode d'évaluation des requêtes sur les documents assujetti à des politiques du contrôle d'accès.

Soient deux alphabets :

- Σ – contenant tous les noms d'éléments,
- Att – composé de noms de tous les attributs

de tous les documents considérés. Par $\mathcal{D}$ on dénote le domaine de toutes les valeurs possibles, de tous les attributs dans Att. En suivant l'idée introduite dans [66], on suppose que les PCDATA — l'information textuelle stockée aux nœuds feuilles des documents XML — sont représentées sous la forme "$text = data$", où $text \in Att$ est un attribut spécifique, et $data$ est le texte considéré. Les requêtes considérées ici sont supposées *positives*, c.–à–d., qu'il n'y a pas de négation sur leur partie de navigation; néanmoins, on autorise la négation dans les filtres, pour comparer des valeurs des attributs. Puisque les requêtes qu'on veut utiliser peuvent questionner ngm

$$\begin{array}{lcl} Q_0 & : & \texttt{/A::}x \mid \texttt{/A::}x\texttt{[}F\texttt{]} \mid Q_0Q_0 \\ Q & : & Q_0 \mid Q_0\texttt{/attribute::}att \mid Q_0\texttt{/attribute::}* \end{array}$$

Table 2.5. Grammaire pour des requêtes avec données

forme $\texttt{/A::}x\texttt{[}L\texttt{]}\texttt{[}G\texttt{]}$, où $\texttt{[}L\texttt{]}$ est un filtre local, et $\texttt{[}G\texttt{]}$ contient les axes de navigation (éventuels). Une requête élémentaire $\texttt{/A::}x\texttt{[}L\texttt{]}$ qui ne contient que le filtre local, sera appelée *atomique*. La requête élémentaire $\texttt{/A::}x\texttt{[}L\texttt{]}\texttt{[}G\texttt{]}$ dont le filtre $\texttt{[}G\texttt{]}$ contient des axes de navigation, sera appelée *non–atomique*. L'évaluation d'une requête de la forme $Q = Q_0\texttt{/attribute::}att$, sur un document t, nous permet de voir les valeurs d'attribut att stockés aux nœuds de t, qui vérifient le chemin d'accès Q_0. Pour voir les valeurs de tous les attributs aux nœuds vérifiant le chemin d'accès définie par Q_0, on emploiera la requête $Q = Q_0\texttt{/attribute::}*$.

Chapitre 3

Évaluation de requêtes et contrôle d'accès

Nous présentons maintenant la première de nos approches, qui permet d'évaluer des requêtes positives de XPath sur les documents XML arborescents assujettis à une politique du contrôle d'accès. Cette approche est basée sur des systèmes de transitions, encodés à l'aide des clauses de Horn. Une telle vue clausale permet d'incorporer d'une façon naturelle des politiques du contrôle d'accès. Ce chapitre est structuré comme suit : Nous commençons (Section 3.1) par introduire les notions nécessaires et les notations. Les Sections 3.2–3.4 sont entièrement consacrées à la construction des systèmes de transitions servant à évaluer les requêtes. Dans la Section 3.5, nous présentons une stratégie qui garantit (Proposition 3.1) que le temps d'évaluation d'une requête Q sur un document t est linéaire par rapport à la taille de Q et le nombre d'arêtes sur t. Dans la Section 3.6 nous expliquons comment évaluer les requêtes en présence d'un mécanisme du contrôle d'accès également basé sur les clauses. Finalement (Section 3.7) nous montrons comment encoder à l'aide des clauses les méthodes classiques traitant du contrôle d'accès aux documents XML (par exemple : l'attribution des clefs). Notons que l'approche présentée dans ce chapitre a été publiée dans [31].

3.1. Préliminaires

L'approche présentée dans ce chapitre sert à évaluer les requêtes positives de XPath, sur les documents arborescents du type XML. L'objectif est d'introduire une méthode d'évaluation de requêtes, à laquelle on pourra incorporer facilement un mécanisme du contrôle d'accès. Nous avons basé notre approche sur des systèmes de transitions définis à l'aide de clauses de Horn contraintes; les techniques de la résolution clausale sont appliquées pour évaluer les requêtes. Ensuite, nous montrons que des politiques du contrôle d'accès, exprimées également en termes de clauses, peuvent être naturellement intégrées dans cette méthode d'évaluation. En outre, il est possible de couvrir, avec notre formalisme clausal, bien d'autres approches abordant le problème du contrôle d'accès.

Soit un document XML t. Rappelons qu'on note par $Root_t$ la racine fictive du document t. Pour tout nœud u de t, on note par $Data_t(u)$ la donnée stockée en u (c.–à–d., toutes les paires $att = val$ correspondant à l'élément de t représenté par u). On pose $t(u) = (name_t(u), Data_t(u))$, où $name_t(u)$ est le nom d'élément représenté par u sur t. A tout axe de base `axis` de XPath, on associe un prédicat booléen `Fin-axis()`, qui s'évalue en *vrai* à un nœud u de t si et seulement si, il n'existe aucun nœud v, tel que u `dir-axis` v soit satisfait sur t. Par exemple,

- `Fin-child(`u`)` et `Fin-descendant(`u`)` sont *vrais* si et seulement si u est une feuille de t;
- `Fin-following-sibling(`u`)` est *vrai* si et seulement si u n'a aucun frère suivant sur t.

Par convention, on suppose que `Fin-self(`u`)` est *vrai*, pour tout nœud u de t.

On considère ici les requêtes que l'on peut générer à partir de la grammaire donnée dans la Table 2.5 (Section 2.5). Soit une telle requête $Q = /C_1/C_2/\ldots/C_n$. On rappelle que pour tout $i \in \{1,\ldots,n\}$, la requête élémentaire $/C_i$ est soit de la forme $/C_i =$`/axis`$_i$`::`σ_i`[`L_i`]`, soit $/C_i =$`/axis`$_i$`::`σ_i`[`L_i`][`F_i`]`, où `[`L_i`]` est le filtre local (ne contenant pas d'expressions de navigation), et `[`F_i`]` est le filtre qui contient les axes de navigation. Pour tout $1 \leq i \leq n$, on construit un système de transitions (ETS — elementary transition system) S_i, à l'aide duquel on évalue la requête élémentaire $/C_i$. Le système de transitions S_Q, évaluant la requête Q tout entière, est alors défini comme une concaténation des systèmes élémentaires S_i, $1 \leq i \leq n$.

On commence par présenter la construction des ETS S_i correspondant à une requête élémentaire $/C_i$ atomique (Section 3.2) et non–atomique (Section 3.3). Ensuite, dans la Section 3.4 on montre comment évaluer une requête donnée Q, à l'aide du système S_Q. Pour simplifier la notation, on suppose que les filtres de Q ne contiennent pas de disjonctions ni conjonctions. Les requêtes ayant des filtres disjonctives ou conjonctives peuvent être évaluées sans problème, en utilisant l'union ou l'intersection d'ensembles des réponses, comme dans la méthode présentée dans le Chapitre 4.

3.2. Système de transitions pour une requête élémentaire atomique

Dans cette section, on considère le cas d'une requête élémentaire atomique qui est de la forme $/C_i =$`/axis`$_i$`::`σ_i`[`L_i`]`, où `[`L_i`]` est un filtre local (sans axes de navigation). Le système de transitions (ETS — *elementary transition system*), pour une telle requête élémentaire atomique, est un système noté S_i, dont l'ensemble des états est $States_i = \{init_i, ok_i, fail_i\}$, et dont les transitions sont définies par des clauses $t1,\ldots,t6$ ci–dessous, où $\alpha_i = \sigma_i$`[`L_i`]`, et $\overline{\alpha_i} = \overline{\sigma_i\texttt{[}L_i\texttt{]}}$ représente le complémentaire de σ_i`[`L_i`]` :

- Si `axis`$_i \in$ `{self, child, parent}` :
  $t1$. $\langle ok_i, v\rangle \leftarrow \langle init_i, u\rangle$, si $(u$ `dir-axis`$_i\, v)$, $(t(v) \models \alpha_i)$
  $t2$. $\langle fail_i, v\rangle \leftarrow \langle init_i, u\rangle$, si $(u$ `dir-axis`$_i\, v)$, $(t(v) \models \overline{\alpha_i})$
- Si `axis`$_i \notin$ `{self, child, parent}` :
  $t3$. $\langle ok_i, v\rangle \leftarrow \langle init_i, u\rangle$, si $(u$ `dir-axis`$_i\, v)$, $(t(v) \models \alpha_i)$,
  $t4$. $\langle fail_i, v\rangle \leftarrow \langle init_i, u\rangle$, si $(u$ `dir-axis`$_i\, v)$, $(t(v) \models \overline{\alpha_i})$,
  $t5$. $\langle fail_i, v\rangle \leftarrow \langle fail_i, u\rangle$, si $(u$ `dir-axis`$_i\, v)$, $(t(v) \models \overline{\alpha_i})$,
  $t6$. $\langle ok_i, v\rangle \leftarrow \langle fail_i, u\rangle$, si $(u$ `dir-axis`$_i\, v)$, $(t(v) \models \alpha_i)$.

Soit t un document (arbre XML) donné. Le rôle du système S_i est de sélectionner (en assignant l'état sélectionnant ok_i) tous les nœuds du t, qui répondent à la partie $/C_1/\ldots/C_i$ de la requête Q considérée. L'assignation des états du système S_i aux nœuds de t se fait par l'intermédiaire d'un run de S_i sur t. Le *run* du ETS S_i sur

l'arbre t est une fonction M_i: $Nodes_t \to \mathcal{P}(States_i)$ qui est conforme aux règles de transition $t1-t6$, et qui satisfait en plus les deux règles présentées plus bas, où v est un nœud de t, et $\langle q, v\rangle$ veut dire $q \in M_i(v)$:

1a. $\langle init_i, v\rangle \leftarrow v = Root_t$, si $i = 1$;
$\langle init_i, v\rangle \leftarrow \langle ok_{i-1}, v\rangle$, si $i > 1$;

2a. $\langle init_i, v\rangle \leftarrow \langle ok_i, v\rangle$, si $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$.

Pour chaque nœud v de t, l'ensemble $M_i(v)$ contient *tous les états* assignés à v durant la traversée du document t. Les ensembles $M_i(v)$, $1 \leq i \leq n$, sont construits récursivement par rapport à i, les uns après les autres. Tout d'abord, en utilisant la transition 1a, on ajoute l'état $init_i$ à chaque ensemble $M_i(v)$, tel que : $v = Root_t$ si $i = 1$, ou v est un nœud sélectionné par le système S_{i-1} lorsque $i > 1$. Ensuite, la construction continue pour chaque v de la façon suivante : le run du système S_i traverse le document t dans le sens défini par $\texttt{axis}_i$, en utilisant les clauses $t1, \ldots, t6$. Ces clauses permettent de sélectionner les nœuds où les données sont consistantes avec le test $\alpha_i = \sigma_i[L_i]$. Si le run M_i utilise une transition de la forme

$$\langle q, v\rangle \leftarrow \langle p, u\rangle, \quad \text{si } (u\, \texttt{dir-axis}_i\, v),\ (t(v) \models \gamma),$$

on dira qu'il se *déplace* (ou effectue un *mouvement*) à partir du nœud u vers le nœud v. La sémantique d'un tel mouvement est la suivante : le système S_i se trouve dans l'état p au nœud courant u. En utilisant la clause mentionnée, le run de S_i assigne au nœud v satisfaisant $((u\, \texttt{dir-axis}_i\, v)$ et $t(v) \models \gamma)$ l'état q (c.–à–d., ajoute l'état q à l'ensemble $M_i(v)$). Si la donnée au nœud v satisfait $(t(v) \models \alpha_i)$, alors l'état q assigné à v est $q = ok_i$, et le nœud v est sélectionné par S_i; sinon l'état assigné à v est $fail_i$. Maintenant, le système S_i se trouve au nœud v qui constitue à présent le nœud de départ (appelé nœud *courant*) pour un déplacement ultérieur. Remarquons que si $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$, la transition 2$a$ est utilisée pour continuer la recherche d'autres nœuds satisfaisant α_i, à partir du nœud v déjà sélectionné (c.–à–d., tel que $ok_i \in M_i(v)$).

Notons que le run du système S_i est *déterministe* car à chaque instant il existe une seule transition applicable. En effet, si le système S_i se déplace d'un nœud u — tel que le dernier état ajouté à l'ensemble $M_i(u)$ est p — vers un nœud v satisfaisant $(u\, \texttt{dir-axis}_i\, v$ et $t(v) \models \gamma)$, alors il existe une seule transition applicable :

$$\langle q, v\rangle \leftarrow \langle p, u\rangle, \quad \text{si } (u\, \texttt{dir-axis}_i\, v),\ (t(v) \models \gamma).$$

Par suite, on ajoute à l'ensemble $M_i(v)$, l'état q, tel que :

$$q = \begin{cases} ok_i, & \text{si } \gamma = \alpha_i \\ fail_i, & \text{si } \gamma = \overline{\alpha_i}. \end{cases}$$

3.3. Système de transitions pour une requête élémentaire non–atomique

Considérons maintenant le cas plus général, celui d'une requête élémentaire non–atomique $/C_i = /\texttt{axis}_i^0 :: \sigma_i^0[L_i^0]\,[F_i]$, contenant un filtre non–local $[F_i]$. Une telle requête peut être écrite sous la forme

$$/C_i = step_i^0[step_i^1[step_i^2[\ldots[step_i^{k(i)}]\ldots],$$

pour un certain entier naturel $k(i)$, où chaque étape $step_i^p = \texttt{axis}_i^p::\sigma_i^p[L_i^p]$, pour $0 \leq p \leq k(i)$, est *atomique* (c.–à–d., $[L_i^p]$ ne contient aucun axe de navigation). Pour tout $0 \leq p \leq k(i)$, on définit d'abord un système de transitions S_i^p appelé STS (*single step transition system*), correspondant à l'étape $step_i^p$. L'ETS S_i correspondant à la requête élémentaire $/C_i$ est alors défini comme une concaténation des STSs S_i^p, pour $0 \leq p \leq k(i)$.

Pour tout $p \in \{0, \ldots, k(i)\}$, notons par α_{ip} la donnée $\sigma_i^p[L_i^p]$, et par $\overline{\alpha_{ip}}$ son complémentaire $\overline{\sigma_i^p[L_i^p]}$. La construction du STS S_i^p dépend du rôle joué par l'étape de localisation $step_i^p = \texttt{axis}_i^p::\sigma_i^p[L_i^p]$ correspondante. Il y a trois types d'étapes de localisation :

- $/step_i^0[$
  Cette étape correspond à la partie de navigation de la requête élémentaire $/C_i$, qui est suivie par un filtre non–local $[F_i]$. L'ensemble des états du STS S_i^0 correspondant est
  $$States_i^0 = \{init_i^0, fail_i^0, ok_i^0[-]\},$$
  et son ensemble des transitions est obtenu à partir de clauses $t1 - t6$ de la Section 3.2, en remplaçant les états $init_i$, $fail_i$, ok_i respectivement par $init_i^0$, $fail_i^0$, $ok_i^0[-]$.
- $[step_i^p[$
  Dans ce cas $p \in \{1, \ldots, k(i) - 1\}$, et cette étape correspond à un filtre non–local suivi par un autre filtre non–local. L'ensemble des états du STS S_i^p correspondant est
  $$States_i^p = \{init_i^p[-], fail_i^p[-], fail_i^p[\bot], ok_i^p[-]\},$$
  et l'ensemble de ses transitions est obtenu à partir de clauses $t1 - t6$ de la Section 3.2, en remplaçant :
  — dans les clauses $t1$ et $t2$ les états $init_i$, $fail_i$, ok_i respectivement par $init_i^p[-]$, $fail_i^p[\bot]$, $ok_i^p[-]$,
  — dans les clauses $t3 - t6$ les états $init_i$, $fail_i$, ok_i respectivement par $init_i^p[-]$, $fail_i^p[-]$, $ok_i^p[-]$.
- $[step_i^{k(i)}]$
  Cette étape correspond au dernier filtre non–local $[step_i^{k(i)}]$ de la requête élémentaire $/C_i$ considérée. L'ensemble des états du STS $S_i^{k(i)}$ est défini par :
  $$States_i^{k(i)} = \{init_i^{k(i)}[-], fail_i^{k(i)}[-], fail_i^{k(i)}[\bot], ok_i^{k(i)}[\top]\},$$
  et celui de ses transitions est obtenu à partir de clauses $t1 - t6$ de la Section 3.2, en remplaçant :
  — dans les clauses $t1 - t2$, les états $init_i$, $fail_i$, ok_i respectivement par $init_i^{k(i)}[-]$, $fail_i^{k(i)}[\bot]$, $ok_i^{k(i)}[\top]$,
  — dans les clauses $t3 - t6$, les états $init_i$, $fail_i$, ok_i respectivement par $init_i^{k(i)}[-]$, $fail_i^{k(i)}[-]$, $ok_i^{k(i)}[\top]$.

Les transitions du STS S_i^0 servent à trouver des nœuds *potentiellement sélectionnés* par $/C_1/\ldots/C_i$, c.–à–d., des nœuds v, tels que $t(v) \models \sigma_i^0[L_i^0]$, et tels qu'il existe un nœud u sélectionné par la partie $/C_1/\ldots/C_{i-1}$ satisfaisant $u\,\texttt{dir-axis}_i^0\,v$. Le rôle des systèmes S_i^p, pour $1 \leq p \leq k(i)$, est par suite de *valider* ces sélections potentielles de S_i^0 : parmi des nœuds sélectionnés par S_i^0 ils retiendront seulement ceux pour lesquels le filtre $[F_i]$ s'évalue en *vrai*.

Après avoir construit les systèmes S_i^p pour $0 \leq p \leq k(i)$, on peut définir l'ETS S_i correspondant à la requête élémentaire $/C_i = /step_i^0[step_i^1[step_i^2[\ldots[step_i^{k(i)}]\ldots]$ qui contient un filtre non–local $[F_i]=[step_i^1[step_i^2[\ldots[step_i^{k(i)}]\ldots]$. L'ensemble des états de ce ETS S_i est défini par :

$$States_i = \{init_i^0, fail_i^0\} \cup \bigcup_{p=1}^{k(i)} \{init_i^p[\gamma], fail_i^p[\gamma] \mid \gamma \in \{-, \bot, \top\}\} \cup$$
$$\{ok_i^{k(i)}[\top]\} \cup \bigcup_{p=0}^{k(i)-1} \{ok_i^p[\gamma] \mid \gamma \in \{-, \bot, \top\}\},$$

et son ensemble des transitions est l'union des ensembles des transitions de tous les S_i^p, pour $0 \leq p \leq k(i)$.

Étant donné un document t, on définit le *run* d'un tel ETS S_i sur t comme une fonction $M_i\colon Nodes_t \to \mathcal{P}(States_i)$, qui est conforme avec les transitions de S_i, et avec les transitions $1-11$ données plus bas. Avant de présenter les transitions $1-11$, nous introduisons quelques notations. On considère une logique à trois valeurs $\{1, 0, \omega\}$, telles que $0 < \omega < 1$, où le symbole ω représente la valeur indéfinie. On pose : $\overline{1} = 0$, $\overline{0} = 1$, $\overline{\omega} = \omega$. Pour tout $p \in \{0, \ldots, k(i)\}$, on définit ensuite un prédicat unaire Ω_i^p, pouvant avoir trois valeurs 1, 0, ou ω. Ce prédicat va être évalué récursivement, à chaque nœud u du document t, de la façon suivante :

- Cas où $\texttt{Fin-axis}_i^p(u)$ est *vrai* :
  - $\Omega_i^p(u) = 1$ ssi $M_i(u)$ contient un état de la forme $q_i^p[\top]$,
  - $\Omega_i^p(u) = 0$ ssi $M_i(u)$ contient un état de la forme $q_i^p[\bot]$,
  - $\Omega_i^p(u) = \omega$ ailleurs.
- Cas où $\texttt{Fin-axis}_i^p(u)$ est *faux* :
  - $\Omega_i^p(u) = Sup_v\{\Omega_i^p(v) \mid u\ \texttt{dir-axis}_i^p\ v\}$

Voici les transitions qui doivent être satisfaites par le run M_i. Le symbole v représente un nœud du document t, $p \in \{0, \ldots, k(i)\}$, la notation $\langle s, v\rangle$ est utilisée pour dire que l'état s est dans l'ensemble $M_i(v)$, et $q \in \{init, fail, ok\}$:

1. règle de passage entre S_{i-1} et S_i :
   $\langle init_i^0, v\rangle \leftarrow v = Root_t$, si $i = 1$,
   $\langle init_i^0, v\rangle \leftarrow \langle ok_{i-1}, v\rangle$, si $i > 1$
2. règle de passage entre S_i^p et S_i^{p+1} :
   $\langle init_i^{p+1}[-], v\rangle \leftarrow \langle ok_i^p[-], v\rangle$, si $p \leq k(i) - 1$
3. règle de propagation de $[\top]$ tout au long du chemin traversé par S_i :
   $\langle q_i^p[\top], v\rangle \leftarrow \langle q_i^p[-], v\rangle, \Omega_i^p(v)$, pour $p \geq 1$
4. signaler au système S_i^{p-1} que l'étape $step_i^p$ est satisfaite en v :
   $\langle ok_i^{p-1}[\top], v\rangle \leftarrow \langle ok_i^{p-1}[-], v\rangle, \langle init_i^p[\top], v\rangle$

5. validation définitive de la sélection potentielle en v :
   $\langle ok_i, v\rangle \leftarrow \langle ok_i^0[\top], v\rangle$
6. continuer le run M_i à partir d'un nœud v sélectionné :
   $\langle init_i^0, v\rangle \leftarrow \langle ok_i, v\rangle$, si $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$
. l'étape $step_i^p$ n'est pas validée sur la branche courante :
   $\langle fail_i^p[\bot], v\rangle \leftarrow \langle fail_i^p[-], v\rangle$, $\texttt{Fin-axis}_i^p(v)$, si $p \geq 1$
8. règle de propagation de $[\bot]$ tout au long du chemin traversé par S_i :
   $\langle q_i^p[\bot], v\rangle \leftarrow \langle q_i^p[-], v\rangle, \overline{\Omega_i^p(v)}$, si $p \geq 1$
9. signaler au système S_i^{p-1}, que l'étape $step_i^p$ n'est pas satisfaite en v :
   $\langle ok_i^{p-1}[\bot], v\rangle \leftarrow \langle ok_i^{p-1}[-], v\rangle, \langle init_i^p[\bot], v\rangle$, si $\texttt{axis}_i^{p-1} \in \{\texttt{self, child, parent}\}$
10. si l'étape $step_i^p$ n'est pas satisfaite au nœud v, alors poursuivre avec $step_i^{p-1}$:
    $\langle init_i^{p-1}[-], v\rangle \leftarrow \langle ok_i^{p-1}[-], v\rangle, \langle init_i^p[\bot], v\rangle$, si $\texttt{axis}_i^{p-1} \notin \{\texttt{self, child, parent}\}$
11. continuer le run M_i à partir d'un nœud v où la sélection potentielle était rejetée :
    $\langle init_i^0, v\rangle \leftarrow \langle ok$

7, 8 et 10 (avec $p := p + 1$), et assigne l'état $init_i^p[-]$ au nœud v_i^p. Par suite, le run M_i peut continuer la recherche d'autres nœuds où la donnée valide $\sigma_i^p[L_i^p]$.

• **Une sélection potentielle de v_i^0 validée :**
Si l'état $ok_i^{k(i)}[\top]$ est atteint par le run M_i, les clauses 3, 4 sont utilisées pour propager le *symbole de validation* $[\top]$ à tout nœud u traversé par M_i. En particulier, l'état $ok_i^0[\top]$ est ajouté à l'ensemble $M_i(v_i^0)$, et grâce à la clause 5 l'état sélectionnant ok_i est atteint au nœud v_i^0 ($ok_i \in M_i(v_i^0)$).

• **Une sélection potentielle de v_i^0 rejetée :**
La sélection potentielle du nœud v_i^0 ne peut pas être validée, si le filtre $[F_i]$ s'évalue en *faux* en v_i^0. Dans ce cas, le *symbole d'échec* $[\bot]$ est propagé en utilisant les clauses 7–9. En particulier, l'état $ok_i^0[\bot]$ est ajouté à l'ensemble $M_i(v_i^0)$, et par conséquent le nœud v_i^0 ne peut pas être sélectionné.

Après avoir validé ou rejeté la sélection potentielle du nœud v_i^0, et si en plus $\texttt{axis}_i \notin \{\texttt{self, child, parent}\}$, les clauses 6 et 11 permettent de poursuivre le run M_i à partir de v_i^0, pour rechercher d'autres nœuds répondant à $/C_1/\ldots/C_i$. On illustre l'évaluation d'une requête élémentaire non–atomique dans l'exemple qui suit.

Exemple 3.1. *Il s'agit d'évaluer la requête* $Q = \texttt{/descendant::}a\texttt{[ancestor::}b\texttt{]}$ *sur le document t représenté sur la Figure 3.1. Les nœuds de t sont dénotés par leurs positions. La Figure 3.2 représente le run* M_1 *du ETS* S_1*, qui évalue la requête élémentaire* $/C_1 = \texttt{/descendant::}a\texttt{[ancestor::}b\texttt{]}$*. Tout d'abord, on considère l'étape de navigation de* $/C_1$*, c.–à–d.,* $step_1^0 = \texttt{descendant::}a$*. En commençant par la racine fictive* $Root_t$ *de t, le run* M_i *parcourt le document t du haut vers le bas (en suivant l'axe* `descendant`*), conformément à la résolution clausale suivante :*

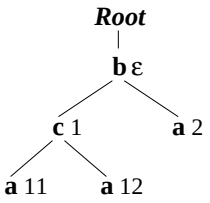


Figure 3.1. un document XML t

$$\langle init_1^0, Root\rangle \leftarrow \qquad \langle ok_1^0[-], 2\rangle \leftarrow \langle fail_1^0, \varepsilon\rangle \tag{3.1}$$

$$\langle fail_1^0, \varepsilon\rangle \leftarrow \langle init_1^0, Root\rangle \qquad \langle ok_1^0[-], 11\rangle \leftarrow \langle fail_1^0, 1\rangle \tag{3.2}$$

$$\langle fail_1^0, 1\rangle \leftarrow \langle fail_1^0, \varepsilon\rangle \qquad \langle ok_1^0[-], 12\rangle \leftarrow \langle fail_1^0, 1\rangle. \tag{3.3}$$

Les états représentés sur la Figure 3.2 à gauche, sont alors assignés aux nœuds de t. L'état $ok_1^0[-]$ *aux nœuds* 2, 11, 12 *signifie que ces nœuds sont potentiellement sélectionnés, et pour valider ou rejeter leur sélection il faut évaluer l'étape* $[step_1^1] = \texttt{[ancestor::}b\texttt{]}$*, pour voir s'ils ont un ancêtre b. L'évaluation de cette étape est représentée sur la Figure 3.2 à droite. Tout d'abord, la clause*

$$\langle init_1^1[-], u\rangle \leftarrow \langle ok_1^0[-], u\rangle \tag{3.4}$$

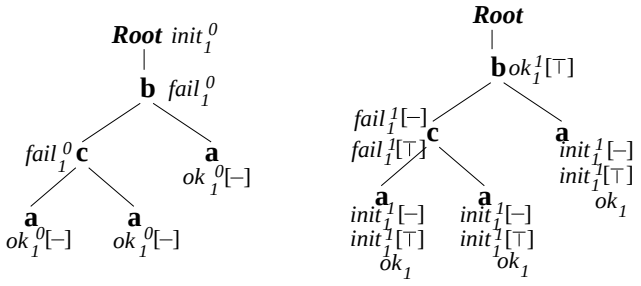


Figure 3.2. Évaluation de `/descendant::`a`[ancestor::`b`]` sur t

est utilisée, aux nœuds $u = 2, 11, 12$. *Ensuite, le run* M_i *utilise les transitions suivantes du STS* S_1^1 *:*

$$\langle fail_1^1[-], 1\rangle \leftarrow \langle init_1^1[-], 11\rangle, \qquad \langle ok_1^1[\top], \varepsilon\rangle \leftarrow \langle fail_1^1[-], 1\rangle, \tag{3.5}$$

$$\langle fail_1^1[-], 1\rangle \leftarrow \langle init_1^1[-], 12\rangle, \qquad \langle ok_1^1[\top], \varepsilon\rangle \leftarrow \langle init_1^1[-], 2\rangle, \tag{3.6}$$

Maintenant, en utilisant la clause 3 *de la définition du run,* M_i *propage le symbole* $[\top]$ *(satisfaction de filtre* `[`$step_1^1$`=[ancestor::`b`]`*) à partir de* ε *aux nœuds* 1, 2, *et ensuite à partir de* 1 *au* 11 *et* 12. *Finalement, grâce à la clause :*

$$\langle ok_1, u\rangle \leftarrow \langle ok_1^0[\top], u\rangle, \tag{3. }$$

l'état sélectionnant ok_1 *est assigné aux nœuds* $u = 11$, 12 *et* 2, *qui constituent la REPONSE à* Q *sur* t.

3.4. Système de transitions pour une requête quelconque

Soient une requête $Q = /C_1/C_2/\ldots/C_n$, et les ETSs S_i, pour $1 \leq i \leq n$, définis dans les deux sections précédentes. On construira maintenant un système de transitions (ST pour abréger) S_Q servant à évaluer Q. L'ensemble des états du système S_Q est défini par

$$States_Q = \bigcup_{i=1}^{n} States_i,$$

et son ensemble des transitions est l'union de tous les ensembles des transitions des ETSs S_i, pour $1 \leq i \leq n$. Pour tout $1 \leq i \leq n$, soit M_i le run du ETS S_i sur un document t donné. Le run M_Q du système S_Q sur t, est défini comme une fonction M_Q: $Nodes_t \rightarrow \mathcal{P}(States_Q)$, où $M_Q = \bigcup_{i=1}^{n} M_i$, qui satisfait deux transitions supplémentaires :

12. $\langle init_1, u\rangle \leftarrow \langle ok_n, u\rangle$, si $\texttt{axis}_1 \notin \{\texttt{self}, \texttt{child}, \texttt{parent}\}$
13. $\langle init_1^0, u\rangle \leftarrow \langle ok_n, u\rangle$, si $\texttt{axis}_1^0 \notin \{\texttt{self}, \texttt{child}, \texttt{parent}\}$.

Un nœud u de t constitue une réponse à la requête Q, ssi $ok_n \in M_Q(u)$. Grâce aux transitions 12 et 13, on peut recommencer l'évaluation de la requête Q à partir d'un nœud déjà sélectionné répondant à Q (bien évidemment, seulement dans le cas où

l'axe de navigation de $/C_1$ n'est ni `self`, ni `child`, ni `parent`). Par suite on trouve exactement *toutes* les réponses à Q sur t.

Notons aussi que, comme dans le cas des ETSs, le run M_Q du système S_Q sur un document t, assigne à chaque nœud u de t, un ensemble $M_Q(u) \subset States_Q$. Cet ensemble est construit d'une façon incrémentale. Le run M_Q commence à la racine de t, ensuite il traverse le document t en suivant les axes des différentes étapes de localisation de Q. Puisque le run M_Q est défini comme l'union des runs M_i des ETSs S_i qui sont déterministes, le ST S_Q lui–même est aussi *déterministe* : il y a toujours au plus une transition applicable pour se déplacer entre deux nœuds donnés.

3.5. Stratégie linéaire d'évaluation d'une requête

On verra maintenant qu'en utilisant l'approche représentée dans ce chapitre, le temps d'évaluation d'une requête Q sur un document t, est linéaire par rapport au nombre d'étapes (atomiques) de localisation de Q et par rapport au nombre d'arêtes de t. Pour cela on introduit une stratégie qui guide la construction du run M_Q, et qui permet à M_Q de *ne traverser chaque nœud de t qu'un nombre constant de fois*. Cette stratégie, appelée par la suite *stratégie linéaire*, est basée sur le concept suivant : *l'évaluation de chaque étape de localisation* $step_i^p$ *(resp.* $step_i$ *dans le cas atomique) à tout nœud u de t, n'est faite qu'une seule fois*. La stratégie mentionnée est formalisée dans la preuve de la proposition ci–dessous :

Proposition 3.1. *La complexité d'évaluation d'une requête XPath Q, sur un document XML donné t, en utilisant le run* M_Q *du système de transition* S_Q*, est linéaire par rapport au nombre d'étapes atomiques de Q et au nombre d'arêtes de t.*

Preuve. La notation utilisée ici est la même que dans les sections précédentes : n est la profondeur (le nombre de requêtes élémentaires) de Q, $i \in \{1, \ldots, n\}$, $k(i)$ est le nombre d'étapes de localisation de la requête élémentaire $/C_i$, et $p \in \{0, \ldots, k(i)\}$. Pour chaque paire d'indices i, p, considérons $step_i^p$ et son axe $\texttt{axis}_i^p$ correspondant. A chaque paire de nœuds u, v de t, tels que u $\texttt{dir-axis}_i^p$ v soit satisfait sur t, ou $u = v$, on associe un ensemble $S_i^p(u, v)$ composé initialement de toutes les transitions du système S_i^p.
Pendant la construction du run M_Q, les ensembles $S_i^p(u, v)$ contiendront les transitions qui peuvent être encore utilisées par M_Q pour passer du nœud u vers le nœud v. Le contenu de l'ensemble $S_i^p(u, v)$ va alors évoluer comme suit :
Considérons deux nœuds u et v, tels que u `dir-A` v soit satisfait sur t. A chaque fois que le run M_Q effectue un déplacement de u vers v, en utilisant une transition ρ du système S_i^p, on enlèvera de tous les ensembles $S_i^p(u', v)$, tels que u' `dir-A` v soit satisfait sur t, *toutes* les règles de transition de S_i^p, ayant le même littéral de tête que ρ. Une telle évolution des ensembles $S_i^p(u, v)$ est justifiée, car le système S_Q est déterministe. Par suite, si le run M_Q se déplace du u vers v en utilisant le système S_i^p, il ne peut utiliser que des transitions qui sont toujours présentes dans l'ensemble $S_i^p(u, v)$. En particulier, aucune arête de t n'est traversée plus qu'une fois par la même transition de S_Q.

□

Illustrons la stratégie linéaire sur un exemple :

Exemple 3.2. *On évalue la requête*

$$Q = /\texttt{descendant::}a\texttt{[ancestor::}b\texttt{]/parent::}c,$$

sur l'arbre t de la Figure 3.1.

Le run évaluant la partie $/C_1 = /\texttt{descendant::}a\texttt{[ancestor::}b\texttt{]}$ a été déjà présenté dans l'Exemple 3.1. On montrera ici comment optimiser cette évaluation. Après avoir utilisé les clauses 3.1, 3.2 et 3.3 (voir l'Exemple 3.1), en appliquant la stratégie linéaire, on enlève :

- *de l'ensemble $S_1^0(\varepsilon, 1)$ toutes les clauses de S_1^0 dont la tête est $\langle fail_1^0, 1\rangle$,*
- *de l'ensemble $S_1^0(\varepsilon, 2)$ toutes les clauses dont la tête est $\langle ok_1^0[-], 2\rangle$,*
- *des ensembles $S_1^0(1, 11)$ et $S_1^0(1, 12)$ toutes les clauses dont les têtes sont respectivement $\langle ok_1^0[-], 11\rangle$ et $\langle ok_1^0[-], 12\rangle$.*

Ensuite, l'utilisation de la clause $\langle init_1^1[-], u\rangle \leftarrow \langle ok_1^0[-], u\rangle$ aux nœuds $u = 2, 11, 12$, entraîne la suppression de toutes les clauses ayant la tête de la forme $\langle init_1^1[-], u\rangle$ des tous les ensembles $S_1^1(u, u)$, où $u = 2, 11, 12$.

Pour évaluer le filtre $step_1^1 = \texttt{ancestor::}b$, le run utilise les clauses 3.5 ce qui permet, en utilisant notre stratégie linéaire, de supprimer :

- *des ensembles $S_1^1(11, 1)$ et $S_1^1(12, 1)$, les clauses dont la tête est de la forme $\langle fail_1^1[-], 1\rangle$,*
- *des ensembles $S_1^1(1, \varepsilon)$ et $S_1^1(2, \varepsilon)$ les clauses dont la tête est de la forme $\langle ok_1^1[\top], \varepsilon\rangle$.*

On voit bien que grâce à notre stratégie linéaire, on évite d'utiliser les clauses 3.6 mentionnées dans l'Exemple 3.1, et d'atteindre ainsi les nœuds 1 et ε deux fois. Ensuite, la clause 3 de la définition du run propage le symbole de satisfaction $[\top]$, à partir de ε aux nœuds 1, 2, et à partir de 1 au 11 et 12 (les clauses correspondantes sont enlevées des ensembles $S_1^1(u, v)$ appropriées). Finalement, grâce aux clauses :

$$\langle ok_1, u\rangle \leftarrow \langle ok_1^0[\top], u\rangle, \qquad \langle init_2, u\rangle \leftarrow \langle ok_1, u\rangle,$$

l'état ok_1 est assigné aux nœuds $u = 11$, 12 et 2, et les clauses ayant les têtes de la forme $\langle ok_1, u\rangle$ et $\langle init_2, u\rangle$, sont supprimées des ensembles $S_1^1(u, u)$, où $u = 11, 12, 2$. Ces trois nœuds deviennent par conséquent des nœuds de départ pour l'évaluation de la requête élémentaire suivante $/C_2 = /\texttt{parent::}c$.

L'évaluation de $/C_2$ est représentée sur la Figure 3.3. En commençant par les

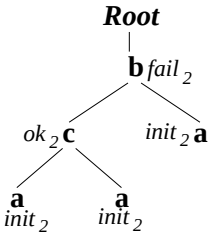


Figure 3.3. Evaluation de $\ldots/\texttt{parent::}c$

nœuds $11, 12$ *and* 2*, le run de* S_Q *traverse le document d'une façon montante (axe* `parent`*). Il utilise la clause*

$$\langle ok_2, 1\rangle \leftarrow \langle init_2, 11\rangle$$

qui permet de sélectionner le nœud à la position 1*. En même temps, les clauses avec la tête* $\langle ok_2, 1\rangle$ *sont enlevées des ensembles* $S_1^1(11, 1)$ *et* $S_1^1(12, 1)$*, ce qui permet de ne pas monter au nœud* 1 *deux fois (à partir de* 11 *et à partir de* 12*). Enfin, pour monter du nœud à la position* 2 *vers la racine* ε*, la transition*

$$\langle fail_2, \varepsilon\rangle \leftarrow \langle init_2, 2\rangle$$

est utilisée, et les clauses ayant la tête $\langle fail_2, \varepsilon\rangle$ *sont supprimées des ensembles* $S_1^1(2, \varepsilon)$ *et* $S_1^1(1, \varepsilon)$*.*

Puisque le premier axe de navigation de Q *est* `descendant`*, on doit poursuivre le run* M_Q *à partir du nœud sélectionné* 1*, pour chercher d'autres nœuds répondant à* Q*. On utilise alors la clause* 13 *:*

$$\langle init_i^0, 1\rangle \leftarrow \langle ok_2, 1\rangle,$$

et on s'aperçoit que notre stratégie linéaire nous assure qu'il n'y a pas d'autres nœuds sur t *qui répondent à* Q*. En effet, les seules clauses utilisables maintenant seraient :*

$$\langle ok_1^0[-], 11\rangle \leftarrow \langle init_1^0, 1\rangle \qquad \langle ok_1^0[-], 12\rangle \leftarrow \langle init_1^0, 1\rangle, \tag{3.8}$$

mais elles ont été enlevées des ensembles $S_1^0(1, 11)$ *et* $S_1^0(1, 12)$ *au début d'évaluation lorsqu'on a atteint des nœuds* 11 *et* 12 *pour la première fois, en utilisant les clauses :*

$$\langle ok_1^0[-], 11\rangle \leftarrow \langle fail_1^0, 1\rangle \qquad \langle ok_1^0[-], 12\rangle \leftarrow \langle fail_1^0, 1\rangle,$$

qui ont les mêmes têtes que les clauses mentionnées 3.8. Par conséquent, le nœud à la position 1 *est la seule réponse à* Q *sur* t*.*

3.6. Évaluation de requêtes sous contrôle d'accès

Nous montrerons maintenant, comment incorporer à notre vue clausale d'évaluation de requêtes, les politiques du contrôle d'accès. Nous modéliserons ces dernières à l'aide des clauses de Horn contraintes. Pour motiver notre approche, nous commençons par un exemple :

Exemple 3.3. *Considérons un document XML représentant une faculté d'une université (Figure 3.4). On suppose que deux catégories d'utilisateurs ont l'accès à ce document — des employés administratifs* Adm*, et académiques* Acad *— mais les représentants de ces deux groupes ne peuvent pas accéder à la même information :*

1. Adm *— un utilisateur administratif ne peut pas accéder à l'information suivante : le nom d'un étudiant (Student.name) et la catégorie (Course.grade) d'un cours que suit ce dernier. Pourtant, il peut consulter séparément ces deux informations;*

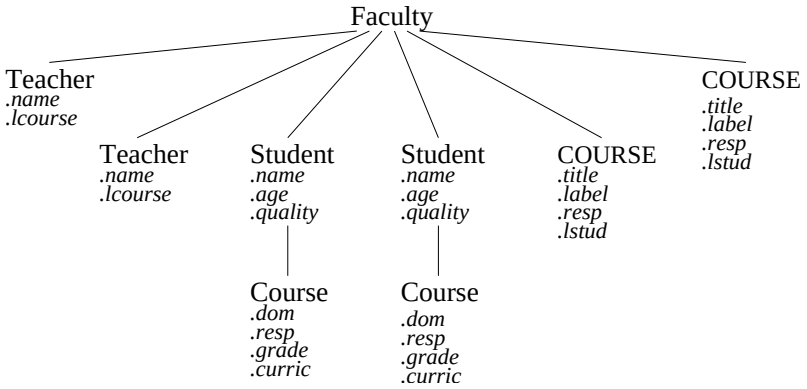


Figure 3.4. XML document représentant une faculté

2. Acad — *un utilisateur académique peut accéder à l'information composée du nom d'un étudiant (Student.name) et la catégorie (Course.grade) d'un de ses cours, si et seulement si l'utilisateur est la personne responsable pour le cours en question (Course.resp).*

On représente les catégories d'utilisateurs Adm *et* Acad *par des prédicats unaires* $Adm(x)$ *et* $Acad(x)$*, qui s'évaluent en* vrai *si et seulement si,* x *est l'identificateur respectivement d'un employé administratif ou académique. Voici comment modéliser à l'aide des clauses purement négatives, la politique du contrôle d'accès mentionnée :*

1. $\leftarrow Adm(*)$, $Student.name$, $Course.grade$,
$[Student.name$ `child` $Course.grade]$

2. $\leftarrow Acad(x)$, $Student.name$, $Course.grade$,
$[Student.name$ `child` $Course.grade,\ Course.resp \neq x]$

La politique du contrôle d'accès pour un utilisateur administratif quelconque $(Adm(*))$ *sera violée, si les informations consultées par ce dernier — peut–être à plusieurs reprises — contiendront le nom d'un étudiant* $Student.name$ *et la catégorie d'un de ses cours* $Course.grade$*, ce qui est exprimé par la clause 1.*

Pendant que l'utilisateur ayant un identificateur id évalue des requêtes sur le document donné, on construira un ensemble $Hist_{id}$*, qui représentera les connaissances acquises par id sur le document* Faculty*. Cet ensemble sera composé des clauses purement positives, et contiendra au moins la clause* $Cat(id) \leftarrow$*, où* Cat *représente la catégorie d'utilisateurs à laquelle appartient id. L'évaluation des requêtes par id restera "sécurisée" tant que l'ensemble* $Hist_{id}$ *sera consistent avec l'ensemble* $\mathcal{P}$ *composé des clauses 1 et 2, modélisant la politique du contrôle d'accès donnée.*

On introduit maintenant les notions qui seront utilisées pour formaliser la méthode présentée dans l'exemple plus haut. Par $User$ on note un ensemble fini d'utilisateurs, et par $Category$ on comprend l'ensemble des utilisateurs d'une même catégorie (par exemple, $Category = Adm$ dans l'Exemple 3.3). On associe à $User$ et $Category$ les prédicats unaires correspondant $User()$ et $Cat()$. Fixons un document t assujetti à une politique du contrôle d'accès $\mathcal{P}$. Les règles de $\mathcal{P}$ sont représentées comme des *clauses de Horn purement négatives*, souvent accompagnées par des con-

traintes exprimant leur portée. Tout littéral apparaissant dans une telle clause est soit un prédicat unaire de la forme $User(x)$ ou $Cat(x)$, soit un symbole propositionnel de la forme $\sigma.a$, où σ est le nom d'un élément sur le document t, et a est le nom d'un attribut associé à σ. La *contrainte de portée* est par définition de la forme $[constr]$, où *constr* est une conjonction finie des expressions $\sigma.a$ `axis` $\rho.b$, ou $\sigma.a$ `dir-axis` $\rho.b$, ou/et $\sigma.a\, op\, val$, où *op* est un opérateur de comparaison ($=, \leq, \geq$ etc.), et *val* est une des valeurs possibles pour l'attribut a.

Exemple 3.4. *Par exemple, la clause*

$$\longleftarrow User(10),\ \sigma.a,\ \rho.b,\ \delta.c, [\sigma.a\ \texttt{child}\ \rho.b,\ \sigma.a\ \texttt{right}\ \delta.c]$$

stipule que l'utilisateur ayant l'identité numéro 10 *n'est pas autorisé à accéder*

dans la contrainte de portée d'une des clauses de $\mathcal{P}$, alors on crée une clause positive

$$[name_t(u').a \texttt{ axis } name_t(v).att] \leftarrow,$$

et on pose :

$Scope(v, Q^{(i)}) =: Scope(v, Q^{(i-1)}) \cup \{[name_t(u').a \texttt{ axis } name_t(v).att] \leftarrow\}$.

2. Pour tout $name_t(v).att$ qui apparaît dans le corps d'une clause de $\mathcal{P}$, tel que $name_t(v).att$ soit consistent avec la donnée $\sigma\texttt{[}L\texttt{]}$ (où l'étape de localisation évaluée à présent par $S_{Q^{(i)}}$ est de la forme $\texttt{axis::}\sigma\texttt{[}L\texttt{]}$), on crée la clause

$$name_t(v).att \leftarrow,$$

et on pose :

$$Access(v, Q^{(i)}) := Access(v, Q^{(i-1)}) \cup \{name_t(v).att \leftarrow\}.$$

3. On définit $Hist(v, Q^{(i)}) := Hist(v, Q^{(i-1)}) \cup Scope(v, Q^{(i)}) \cup Access(v, Q^{(i)})$.

Step 3. On pose $i := i + 1$, et on revient à **Step 1**.

Par $\mathcal{D}_v(t)$ on note l'ensemble des dépendances fonctionnelles (éventuelles) entre les attributs stockés au nœud v d'un document t. On supposera que ces dépendances sont formulées également sous forme de clauses de la forme

$$name_t(v).a_{i_r} \leftarrow name_t(v).a_{i_1}, name_t(v).a_{i_2}, \ldots, name_t(v).a_{i_k},$$

où a_j sont les noms des attributs en v. S'il n'y a pas de telles dépendances, on pose $\mathcal{D}_v(t) := \emptyset$. Notons par $Hist_{id}$ l'ensemble étant l'union de tous les ensembles $Hist_{id}(v, Q^i)$ construits plus haut. En utilisant les ensembles $Hist_{id}$ et $\mathcal{D}_v(t)$, on définit une stratégie d'évaluation qui garantit que l'évaluation d'une requête ne viole pas la politique du contrôle d'accès donnée :

Définition 3.1. (Évaluation sécurisée) *Si une requête donnée Q est évaluée par un utilisateur id, à l'aide du système S_Q, sur un document t assujetti à une politique d'accès $\mathcal{P}$, alors un état sélectionnant peut être assigné à un nœud v de t, si et seulement si*

$$Hist_{id} \cup \mathcal{D}_v(t) \cup \mathcal{P} \not\models \bot.$$

Pour vérifier la condition de la Définition 3.1 on utilise la résolution clausale. Voici certaines règles (parmi bien d'autres) que l'on peut appliquer pour une telle résolution :

- la résolution classique entre des littéraux positifs et négatifs;
- un littéral $\sigma.att$ peut être résolu avec un littéral (de signe opposé) de la forme $\sigma.*$; idem pour des littéraux (ayant des signes opposés) $User(id)$ et $User(*)$, ainsi que $Cat(id)$ et $Cat(*)$ etc.;
- une contrainte de portée $[\sigma.a \texttt{ axis } \rho.b]$ peut être résolu avec une contrainte de portée de la forme $[\rho.b \texttt{ axis}^{-1} \sigma.a]$;
- un littéral négatif $[\sigma.a \texttt{ axis } \rho.b]$ peut être résolu avec un littéral positif de la forme $[\sigma.a \texttt{ dir-axis } \rho.b]$;

- un littéral négatif $[\sigma.a$ `axis` $\rho.b]$ peut être résolu avec un littéral positif de la forme $[\sigma.a$ `axis` $\rho.b,\ \sigma.a \neq \text{`}val'].$

Si on ne veut pas qu'une politique du contrôle d'accès soit violée, il est indispensable de garder la trace de toute information à laquelle un utilisateur a accédé en évaluant ses différentes requêtes, comme le montre l'exemple suivant.

Exemple 3.5. *Soient le document* Faculty *et la politique du contrôle d'accès considérés dans l'Exemple 3.3. Supposons que l'attribut grade stocké à tout nœud Course n'a pas toujours la même valeur, c.–à–d., les étudiants de catégories (grade) différentes peuvent suivre le même cours. Supposons qu'un utilisateur administratif id évalue successivement les trois requêtes suivantes :*

$Q_1 = //Student/@*$

$Q_2 = //Course/@grade$

$Q_3 = //Course[@grade \neq M]$`/parent::`$Student/@name$

Analysons ce qui se passe si l'ensemble $Hist_{id}$ *n'est pas gardé :*

- *En évaluant* Q_1*, l'utilisateur prend connaissance de toutes les données stockées à chaque nœud Student, en particulier les noms de tous les étudiants.*
- *En évaluant* Q_2 *il obtient les valeurs d'attribut grade pour tous les nœuds Course.*
- *La REPONSE à* Q_3 *donne à notre utilisateur les noms de tous les étudiants qui ne sont pas enregistrés dans la catégorie M.*

Il a maintenant toutes les informations suffisantes pour déduire les noms des étudiants enregistrés dans la catégorie M — il suffit de prendre le complément de l'ensemble qui constitue la REPONSE à Q_3 *— ce qui viole la politique du contrôle d'accès imposée dans l'Exemple 3.3 :*

$\leftarrow Adm(*)$, $Student.name$, $Course.grade$,
$[Student.name$ `child` $Course.grade].$

Garder l'ensemble $Hist_{id}$ — la trace d'information à laquelle l'administratif id à accédé — permet d'éviter la violation de la politique du contrôle d'accès donnée. En effet, après avoir évalué les requêtes Q_1 et Q_2, $Hist_{id}$ va contenir les clauses positives suivantes :

$Adm(id) \leftarrow,$
$student.* \leftarrow,$
$course.grade \leftarrow.$

Pendant l'évaluation de la requête Q_3 on y aura ajouté la clause :

$[course.grade$ `parent` $student.name, course.grade \neq M] \leftarrow.$

En particulier, la contrainte de portée $[course.grade \neq M]$ s'évalue en *vrai*. Il n'est pas difficile de remarquer (en utilisant la résolution clausale) que l'ensemble $Hist_{id}$ n'est pas consistent avec la clause

$\leftarrow Adm(*)$, $Student.name$, $Course.grade$,
$[Student.name$ `child` $Course.grade].$

Par conséquent, l'évaluation de Q_3 va être interrompue, et ne fournira aucun nom d'étudiant.

Notre méthode d'évaluation de requêtes sur les documents assujettis à une politique du contrôle d'accès est complète, comme le montre la proposition suivante, dont la preuve découle directement des définitions introduites dans cette section (notamment la Définition 3.1) :

Proposition 3.2. *Soit un document t assujetti à une politique du contrôle d'accès $\mathcal{P}$. Une donnée stockée comme la valeur d'un attribut att au nœud u du t est accessible pour un utilisateur id, si et seulement si $\{name_t(u).att \leftarrow\} \cup Hist_{id} \cup \mathcal{D}_u(t)$ est consistent avec l'ensemble des clauses négatives $\mathcal{P}$.*

3.7. Pouvoir d'expression de la vue clausale pour le contrôle d'accès

Pour finir ce chapitre, on montre comment encoder, en utilisant notre vue clausale, deux méthodes souvent utilisées pour traiter le problème du contrôle d'accès aux documents XML.

I.) Pour couvrir les approches basées sur l'attribution des clefs aux nœuds et/ou leurs attributs, (voir par exemple [2] et [66]), on définit la notion de *clef* K sur un document t, comme une fonction (partielle) K: $Nodes_t \cup Att \rightarrow 2^{User}$, telle que $K(x)$ soit un ensemble fini d'utilisateurs. Voici les clauses encodant l'attribution des clefs :

- Soit u un nœud de t. Si $K(u)$ ne contient pas d'identificateur B, alors l'utilisateur B ne peut accéder au nœud u, ni à aucun de ses descendants v :
  $\leftarrow User(B), u.*\,[B \notin K(u)]$,
  $\leftarrow User(B), v.*\,[B \notin K(u),\ u.*\,\texttt{descendant}\,v.*]$.
- Soit att un attribut associé au nœud u. Si $K(att)$ ne contient pas d'identificateur B, alors l'utilisateur B n'a pas le droit d'accéder à la valeur d'attribut att :
  $\leftarrow User(B), u.att\,[B \notin K(att)]$,

Supposons que l'utilisateur B évalue une requête Q sur un document t assujetti à une politique du contrôle d'accès $\mathcal{P}$ stipulant que $B \notin K(u)$. Si le run du système S_Q atteint le nœud u, on ajoute à $Hist_B$ les clauses $User(B) \leftarrow$, et $u.* \leftarrow$. Ensuite, on ajoutera à $Hist_B$ les clauses $v.* \leftarrow$, et $[u.*\,\texttt{descendant}\,v.*] \leftarrow$, pour tout nœud v descendant de u, qui seront atteints en descendant en dessous de u.

II.) Pour encoder, à l'aide des clauses, la méthode présentée dans [1] (esquissée à la page 9 du présent rapport), on introduit une fonction Acc: $Nodes_t \rightarrow \{-r, -R\}$, qui marque des nœuds qui sont interdits d'accès. Par suite, l'approche du [1] peut être encodée par les clauses suivantes :

- interdiction pour tout utilisateur d'accéder au nœud u et tous ses attributs
  $\leftarrow User(*), u.*\,[Acc(u) = -r]$;
- interdiction pour tout utilisateur d'accéder au nœud u et ses descendants ainsi que leurs attributs
  $\leftarrow User(*), u.*\,[Acc(u) = -R]$,
  $\leftarrow User(*), v.*\,[Acc(u) = -R,\ \ u.*\,\texttt{child}\,v.*]$.

Chapitre 4

Requêtes sur documents compressés

Dans ce chapitre nous présentons une approche qui permet d'évaluer les requêtes positives de Core XPath, sur les documents compressés de type XML, sans nécessiter une décompression préalable. Elle repose sur sept automates de mots, qui correspondent aux sept axes de base de Core XPath. En utilisant cette approche, on peut évaluer les requêtes directement sur les documents partiellement ou totalement compressés. La complexité d'une telle évaluation est linéaire par rapport à la taille de la requête et le nombre d'arêtes du document. Ce chapitre est structuré comme suit : Nous commençons par introduire la notion de trdag (Section 4.1), et celle de grammaire normalisée (Section 4.2), qui nous serviront pour modéliser les documents XML compressés. Dans les Sections 4.3 et 4.4 nous construisons nos sept automates, et présentons notre approche pour les requêtes dites de base. Les résultats sur la complexité de notre méthode se trouvent dans la Section 4.5. La Section 4.6 est consacrée à l'évaluation des requêtes composées : conjonctives, disjonctives et imbriquées. Finalement, dans la Section 4.7, nous adaptons notre approche, pour pouvoir déduire, à partir de la REPONSE à une requête donnée Q sur un document compressé t, la REPONSE à Q sur l'arbre $\widehat{t}$ équivalent à t. Notons que la méthode présentée ici a été publiée dans [30].

4.1. Représentation compressée de documents arborescents

Les volumes de données représentées par des documents XML sont en général très importants. Par conséquent, les arbres XML peuvent occuper beaucoup d'espace de mémoire. De plus, la représentation arborescente n'est pas la plus optimale, car sur un arbre la même information peut être représentée plusieurs fois — par exemple, la première, deuxième et quatrième feuille de l'arbre de la Figure 4.1 représentent la même information a. De nombreuses recherches sur une présentation plus économe des documents XML ont été menées durant la dernière décennie. Différentes méthodes de compression d'arbres XML ont été développées (voir la Section 1.3). Dans ce travail on utilisera les graphes orientés sans cycles *dag*s (de l'anglais *directed acyclic graphs*) pour représenter des documents XML. Notons qu'une telle vision permet d'obtenir un gain exponentiel d'espace de stockage d'un document, par rapport à l'espace nécessaire pour stocker le même document sous forme arborescente. On introduit maintenant la notion de *trdag* — graphe à l'aide duquel on modélisera un document XML compressé ou non compressé.

Définition 4.1. *Soit un alphabet de symboles* Σ. *Un trdag sur* Σ *est un couple* $t = ((Nodes_t, Edges_t), name_t)$, *où* $(Nodes_t, Edges_t)$ *est un dag, ayant un seul*

nœud racine noté $root_t$*, où les arêtes sortant du même nœud sont ordonnées, et* $name_t$: $Nodes_t \to \Sigma$ *est une fonction, assignant à chaque nœud* v *de* t *un nom* $name_t(v) \in \Sigma$.

Remarquons que tout arbre ordonné, étiqueté par des symboles d'un alphabet Σ, est un trdag sur Σ, mais la réciproque est fausse. La Figure 4.1 représente trois trdags sur $\Sigma = \{a, b, f\}$, dont seulement le premier à gauche est un arbre. Notons

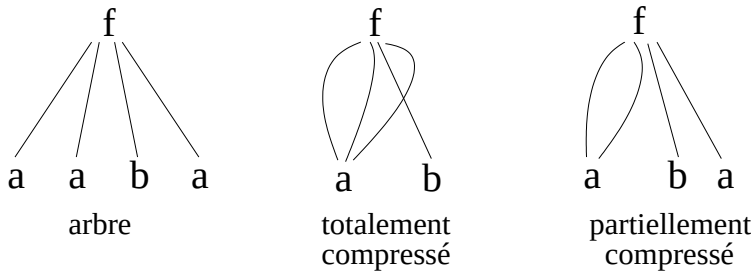


Figure 4.1. Exemples de trdags

aussi, que dans le cas où t n'est pas un arbre, deux différents enfants d'un nœud peuvent être représentés par un seul nœud du t. Par exemple, sur le trdag au milieu de la Figure 4.1, le premier, deuxième ainsi que le quatrième enfant de la racine sont représentés par le même nœud ayant le nom a. Un nœud donné d'un trdag *peut alors représenter son propre frère*, ou *avoir plus qu'un père*.

Soit t un trdag donné. Pour chaque nœud v de t, on définit d'une façon récursive l'ensemble des positions représentées par v sur t, noté $pos_t(v)$:

- si $v = root_t$, alors $pos_t(v) = \{\varepsilon\}$,
- si $v \neq root_t$, alors $pos_t(v) = \{\alpha.i \mid \alpha \in pos_t(w),$ où $w \in Parents_t(v)$ et $\gamma(w) = (u_1, \ldots, u_{i-1}, v, u_{i+1}, \ldots, u_n)\}$.

On définit ensuite l'ensemble des *positions* du trdag t, noté Pos_t, par :

$$Pos_t = \bigcup_{v \in Nodes_t} pos_t(v).$$

La fonction $name_t$ peut être naturellement étendue à l'ensemble Pos_t; il suffit de poser :

$$name_t(\alpha) = name_t(v), \text{ pour tout } \alpha \in pos_t(v).$$

Notons que, si t est un arbre, les ensembles $Nodes_t$ et Pos_t sont en une bijection naturelle, et peuvent être confondus.

Soit un trdag t sur un alphabet Σ. On appelle *arbre équivalent* à t, un arbre sur Σ (unique à isomorphisme près), noté $\widehat{t}$, qui peut être construit d'une façon canonique (voir [35]), en posant :

$$\begin{aligned} &Nodes_{\widehat{t}} = Pos_t, \\ &Edges_{\widehat{t}} = \{(\alpha, \alpha.i) \mid \alpha, \alpha.i \in Pos_t\}, \\ &name_{\widehat{t}}(\alpha) = name_t(\alpha). \end{aligned} \qquad (\widehat{t})$$

On dira alors que t est une *compression* de $\widehat{t}$. On appelle *surjection de compression*, la surjection $\mathbf{c}\colon Nodes_{\widehat{t}} \to Nodes_t$, définie par :

$$\mathbf{c}(\alpha) = v, \text{ si et seulement si } \alpha \in pos_t(v).$$

Par $t|_v$ on désignera le sous–trdag de t, raciné en v. On dira qu'un trdag t est *totalement compressé*, si et seulement si, pour toute paire v, u de nœuds de t : si $v \neq u$, alors les arbres $\widehat{t|_v}$ et $\widehat{t|_u}$ (équivalents aux sous–trdags $t|_v$ et $t|_u$) ne sont pas isomorphes. Tout trdag qui n'est pas totalement compressé sera dit *partiellement compressé*. Sur la Figure 4.1 on présente un trdag totalement compressé (celui au milieu), et deux trdags partiellement compressés (à gauche et à droite).

Comme dans le cas d'une représentation arborescente, on confondra un document XML donné avec le trdag qui le représente.

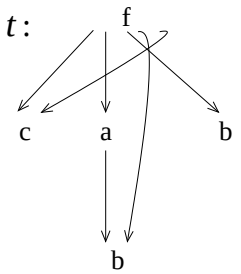
4.2. Trdag vu comme grammaire

On construit maintenant une grammaire associée à un trdag t. On utilisera ensuite le graphe de dépendance de cette grammaire pour évaluer des requêtes XPath sur le document t.

Définition 4.2. *Soit un trdag t donnée. On appelle grammaire normalisée associée à t, une grammaire régulière d'arbres, notée $\mathcal{L}_t$, qui satisfait les conditions suivantes :*

1. *$\mathcal{L}_t$ n'accepte que t,*
2. *le nombre de non–terminaux de $\mathcal{L}_t$ est égal au nombre de nœuds du t,*
3. *pour chaque non–terminal A_i, il existe exactement une production de la forme $A_i \to \sigma(A_{j_1}, \ldots, A_{j_k})$, telle que pour tout $r \in \{1, \ldots, k\}$, on a $i < j_r$; on note alors $Sons(A_i) = \{A_{j_1}, \ldots, A_{j_k}\}$, et $symb_{\mathcal{L}_t}(A_i) = \sigma$.*

Bien évidemment, une telle grammaire peut être produite en temps linéaire par rapport à la taille (nombre d'arêtes) du trdag correspondant, et elle est unique au renommage des non–terminaux près. La Figure 4.2 présente un trdag t et la grammaire normalisée $\mathcal{L}_t$ associée. Les conditions 2 et 3 de la Définition 4.2 impliquent,



$\mathcal{L}_t$:

$A_1 \to f(A_2, A_3, A_4, A_5, A_2)$,
$A_2 \to c$,
$A_3 \to a(A_5)$,
$A_4 \to b$,
$A_5 \to b$

Figure 4.2. Un trdag t et sa grammaire normalisée $\mathcal{L}_t$

en particulier, que l'ensemble des nœuds de t et l'ensemble des non–terminaux de $\mathcal{L}_t$

sont en bijection préservant les symboles. On remarque que la grammaire normalisée de t est une grammaire *straightline*, dans le sens défini dans [1], c.–à–d., chaque non–terminal produit exactement un sous–trdag de t, et il n y a pas de cycles dans la relation de dépendance entre les non–terminaux de $\mathcal{L}_t$.

Soit un trdag t, et sa grammaire normalisée $\mathcal{L}_t$. Notons par n le nombre de nœuds de t. On appelle *graphe de dépendance* de $\mathcal{L}_t$, un graphe

- composé de n nœuds qui portent les noms des non–terminaux $A_1, \ldots, A_n$ de $\mathcal{L}_t$,
- contenant une arête orientée du nœud A_i vers le nœud A_j, si et seulement si $A_j \in Sons(A_i)$.

On étend ce graphe en ajoutant une racine supplémentaire (père de nœud nommé A_1), qui porte le nom A_0; ce nœud supplémentaire représentera la racine fictive du document XML modélisé par t. A chaque nœud v du graphe de dépendance de $\mathcal{L}_t$ ainsi étendu, on ajoute un label $label(v)$ défini comme suit :

$$label(v) = \begin{cases} (Root, -), & \text{si } name(v) = A_0 \\ (symb_{\mathcal{L}_t}(A_i), -), & \text{si } name(v) = A_i \text{ et } i \in \{1, \ldots, n\}. \end{cases}$$

(Notons que le label $label(v)$ utilisé dans ce travail étend la notion de label classique de XML : sa première composante est le nom d'élément représenté par v — donc le label au sens de XML —, et sa deuxième composante (qui pourra être 1, 0, ou $-$) nous indiquera le rôle du nœud v par rapport à la requête évaluée, cf. la sémantique dans la section suivante.) Le graphe labelé ainsi obtenu, noté $\mathcal{D}_t$, sera appelé *rlag de dépendance de* $\mathcal{L}_t$ (rlag étant une abréviation de *rooted labeled acyclic graph*). Le rlag de dépendance $\mathcal{D}_t$ ne contient jamais deux arêtes parallèles. De plus, l'ensemble de nœuds de $\mathcal{D}_t$, qui portent les noms $A_1, \ldots, A_n$, est en bijection avec l'ensemble de nœuds de t; on identifiera souvent donc un nœud de t avec son image sur $\mathcal{D}_t$. Le deuxième graphe sur la Figure 4.3 est le rlag de dépendance de la grammaire normalisée du trdag t représenté à gauche.

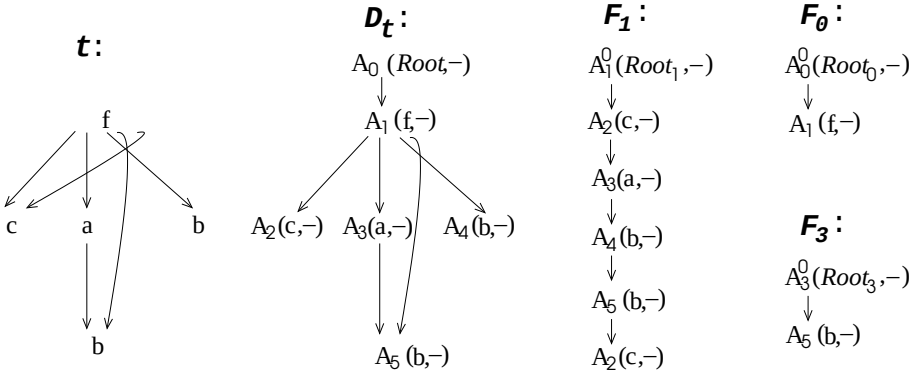


Figure 4.3. trdag t, rlag $\mathcal{D}_t$ et l'ensemble des chaînons correspondants

Soit t un trdag donné, $\mathcal{L}_t$ sa grammaire normalisée, et $\mathcal{D}_t$ le graphe de dépendance correspondant. Remarquons que le rlag $\mathcal{D}_t$ est suffisant pour encoder des relations verticales (père–fils) représentées par t. Néanmoins, faute d'arêtes parallèles, ce n'est

pas une manière appropriée pour représenter les relations horizontales (entre frères) du t. Pour ces dernières, on utilisera une autre famille de graphes, dont voici la construction :
Pour chaque production $A_i \to \sigma(A_{j_1}, \ldots, A_{j_k})$ de $\mathcal{L}_t$, on construit un graphe linéaire contenant j_k nœuds nommés par $A_{j_1}, \ldots, A_{j_k}$, tel que :

- pour tout $l \in \{1, \ldots, k-1\}$, le nœud portant le nom A_{j_l} soit le père du nœud portant le nom $A_{j_{l+1}}$.

On complète ce graphe en ajoutant un nœud racine (fictive) portant le nom A_i^0. Ensuite, à chaque nœud v d'un tel graphe étendu, on définit un label de la façon suivante :

$$label(v) = \begin{cases} (Root_i, -), & \text{si } name(v) = A_i^0 \\ (symb_{\mathcal{L}_t}(A_{j_l}), -), & \text{si } name(v) = A_{j_l}. \end{cases}$$

Le graphe ainsi obtenu sera appelé *i-ème chaînon* de $\mathcal{L}_t$, et sera noté $\mathcal{F}_i$. On notera par $\mathcal{F}_0$ le graphe composée de deux nœuds : la racine portant le nom A_0^0 et le label $label(v) = (Root_0, -)$, et son unique fils portant le nom A_1 et le label $(symb_{\mathcal{L}_t}(A_1), -)$. A droite de la Figure 4.3 on représente les chaînons $\mathcal{F}_0$, $\mathcal{F}_1$ et $\mathcal{F}_3$, correspondants aux productions de la grammaire normalisée $\mathcal{L}_t$ donnée sur la Figure 4.2.

Soit un document t (compressé ou non), et une requête Q. Pour trouver la REPONSE à Q sur t, on utilisera les runs des automates de mots — construits dans la suite de ce chapitre — sur le rlag $\mathcal{D}_t$ et sur les chaînons de la grammaire $\mathcal{L}_t$ associée. L'absence d'arêtes parallèles sur le rlag $\mathcal{D}_t$ implique que ce dernier fournit une représentation très concise du document t (souvent plus concise que t lui même). C'est avantageux pour la complexité de notre approche d'évaluation, qui est (voir la Section 4.5) linéaire par rapport à la taille du graphe sur lequel courent nos automates. Notons aussi que l'approche présentée dans les Sections 4.3–4.6 sert à trouver tous les nœuds qui répondent à la requête Q sur t. Dans la Section 4. , on adaptera cette méthode pour déduire quels sont les nœuds qui répondent à la requête Q sur la représentation arborescente de t.

4.3. Évaluation de requêtes à l'aide des automates de mots

L'approche que l'on présente dans ce chapitre, pour évaluer des requêtes positives de Core XPath sur les documents compressés, est appropriée aux documents représentés sous une forme totalement ou partiellement compressée, ou sous une forme déployée arborescente. Elle couvre ces requêtes en forme standardisée Q_{std} (voir la Section 2.4) que l'on peut générer à l'aide de la grammaire suivante :

$$\begin{array}{lcl} S_{std} & : & \mathit{Root} \mid \texttt{A::}x \mid S_{std}\ \mathit{and}\ S_{std} \mid S_{std}\ \mathit{or}\ S_{std} \\ E_{std} & : & \texttt{A::*[}S_{std}\texttt{]} \mid \texttt{A::*[}E_{std}\texttt{]} \\ Q_{std} & : & \texttt{//*} \mid \texttt{//*[}S_{std}\texttt{]} \mid \texttt{//*[}E_{std}\texttt{]}\,. \end{array}$$

Toutes les requêtes considérées dans ce chapitre seront de ce type. Notons que notre approche peut être naturellement étendue à *toutes* les requêtes standardisées générées à partir de la grammaire donnée dans la Table 2.3, même celles de la forme

`//*[`Y_{std} *conn* Y_{std}`]`. La méthode présentée ici est basée sur sept automates de mots A_Q, correspondant aux sept requêtes, appelées *requêtes de base*, qui sont de la forme $Q =$ `//*[axis::`σ`]`, où `axis` est un axe de base de Core XPath. Soient un document t, et une requête de base $Q =$ `//*[axis::`σ`]`. Pour évaluer Q sur t, on utilisera le run top–down de l'automate A_Q :

- si `axis` est un axe vertical, alors l'automate A_Q courra sur le rlag de dépendance $\mathcal{D}_t$ défini dans la section précédente,
- si `axis` est un axe horizontal, alors l'automate A_Q courra sur l'ensemble de tous les chaînons de la grammaire $\mathcal{L}_t$ associée à t.

Notons que A_Q est non–déterministe (plusieurs runs possibles sur le même rlag), mais grâce à une stratégie dite de *priorité maximale*, on définira ce qu'on appellera *run de priorité maximale* (voir la Définition 4.4) qui sera unique et servira à évaluer la requête Q sur t.

Formalisons maintenant les idées présentées plus haut. Soit un alphabet Σ. On introduit quatre symboles $s, \eta, \top, \top'$ qui n'appartiennent pas à Σ. On appellera *ll–paire* tout couple appartenant à l'ensemble $\{(s,1),(\eta,1),(\eta,0),(\top',1),(\top,1),(\top,0)\}$. Ces couples constituerons par la suite les états de nos automates. Considérons un trdag t sur Σ. Dans la suite de ce chapitre, on notera par $\mathcal{G}_t$:

- le rlag $\mathcal{D}_t$, si `axis` ast un axe vertical,
- tout chaînon $\mathcal{F}$ de $\mathcal{L}_t$, si `axis` est un axe horizontal.

On définit la fonction *llab*: $Nodes_{\mathcal{G}_t} \to \Sigma \cup \{s, \eta\}$, comme la projection de $label(v)$ sur sa première composante :

$$llab(v) = \pi_1(label(v)), \text{ pour tout } v \in Nodes_{\mathcal{G}_t}.$$

Ainsi le label au nœud v est une paire ordonnée, notée $label(v)$, et le llab au nœud v est la première composante de cette paire. Pendant l'évaluation des requêtes composées (Section 4.6), les labels vont évoluer, et les nœuds du rlag $\mathcal{D}_t$ seront relabelés par des ll–paires (parfois plusieurs fois) avant d'obtenir la REPONSE à la requête considérée. Le label $label(v)$ nous informera — conformément à la sémantique de la Table 4.1 — sur le rôle du nœud v par rapport à la partie de la requête que l'on vient d'évaluer : nœud sélectionné ou non (première composante), ayant ou non un descendant sélectionné (deuxième composante).

L'automate A_Q qui permet d'évaluer la requête de base $Q =$ `//*[axis::`σ`]`, où $\sigma \in \Sigma \cup \{*\}$, est définit comme un uplet

$$A_Q = (\Sigma \cup \{s, \eta\}, States_Q, \{init\}, \Delta_Q),$$

où $States_Q \subseteq \{init\} \cup \{(s,1),(\eta,1),(\eta,0),(\top',1),(\top,1),(\top,0)\}$ est l'ensemble des états, *init* est le seul état initial, et Δ_Q est l'ensemble des transitions de la forme

$$(q, \tau) \to q',$$

où $q, q' \in States_Q$, et $\tau \in \Sigma \cup \{*, s, \eta\}$. La composition des ensembles $States_Q$ et Δ_Q varie suivant l'axe `axis` utilisé par la requête $Q =$ `//*[axis::`σ`]`. Les sept automates A_Q, correspondant aux sept requêtes de base $Q =$ `//*[axis::`σ`]` sont représentés sur les Figures 4.4, 4.6–4.11. La construction des automates A_Q obéit aux règles générales suivantes :

Remarque 4.1. *Pour toutes les transitions* $(q, \tau) \to q'$ *de l'automate* A_Q*, on a :*
(i) si $\tau = \sigma$*, alors* $q' \in \{(\top', 1), (\top, 1), (\top, 0)\}$*,*
(ii) si $\tau \neq \sigma$*, alors* $q' \in \{(s, 1), (\eta, 1), (\eta, 0)\}$*,*
(iii) toute transition allant d'une ll–paire avec la deuxième composante 0*, va vers une ll–paire avec la deuxième composante* 0*.*

Soient une requête de base $Q =$ `//*[axis::`σ`]`, l'automate A_Q correspondant, un document t, et le rlag $\mathcal{G}_t$. Voici comment on définit le run de A_Q sur $\mathcal{G}_t$.

Définition 4.3. *Le run de l'automate* A_Q *sur* $\mathcal{G}_t$ *est une fonction* $r\colon Nodes_{\mathcal{G}_t} \to States_Q$*, définie récursivement comme suit : pour tout* $v \in Nodes_{\mathcal{G}_t}$
- *si* $llab(v) = Root$*, ou* $llab(v) = Root_i$*, alors* $r(v) = init$*,*
- *si* v *n'est pas la racine de* $\mathcal{G}_t$*, alors* $r(v)$ *est une ll–paire, telle que pour tout* $w \in Parents(v)$*, la transition* $(r(w), llab(v)) \to r(v)$ *soit dans* Δ_Q*.*

Le run r de l'automate A_Q sur $\mathcal{G}_t$ est ainsi une fonction construite de façon récursive, top–down (suivant tous les chemins racine–feuille de $\mathcal{G}_t$). Elle assigne l'état initial à la racine de $\mathcal{G}_t$, et ensuite une ll–paire $r(v)$ à chaque nœud v de $\mathcal{G}_t$ progressivement. Soit v un nœud de $\mathcal{G}_t$. La deuxième condition de la définition impose, que pour pouvoir définir la valeur $r(v)$, il faut d'abord connaître les valeurs $r(w)$, pour chaque nœud w père de v. De plus, l'état $r(v)$ doit être conforme (par rapport aux transitions de Δ_Q) avec *tous* les états $r(w)$, pour *tout* nœud w père de v. La Remarque 4.1 et la Définition 4.3 impliquent que l'état $r(v)$ est toujours *déterminé par* $llab(v)$, c.–à–d., :
- $r(v) = init$, si $llab(v) = Root$,
- $r(v) \in \{(\top, 0), (\top, 1), (\top', 1)\}$, si $llab(v) = \sigma$,
- $r(v) \in \{(\eta, 0), (\eta, 1), (s, 1)\}$, si $llab(v) \neq \sigma$.

Pour simplifier la notation, on pose $\eta' := s$, et on utilisera souvent la notation $\{(l, 0), (l, 1), (l', 1)\}$, ou $l \in \{\eta, \top\}$, pour désigner le groupe des ll–paires en question.

Soit une requête de base Q, et l'automate A_Q associé. Comme nous allons voir dans les Sections 4.3.1 et 4.3.2, nos automates ne sont pas déterministes, donc à priori il existe plusieurs runs possibles de A_Q sur le même rlag $\mathcal{G}_t$. Pourtant l'automate A_Q est construit pour que sur tout rlag $\mathcal{G}_t$, il existe un seul run de A_Q qui permet d'évaluer la requête Q sur le document t. Ce run assignera à chaque nœud v de $\mathcal{G}_t$, un état dont la sémantique nous informera quelle est la signification de v par rapport à la requête considérée. La sémantique des états de l'automate A_Q, correspondant à la requête de base $Q =$ `//*[axis::`σ`]`, est présentée dans la Table 4.1. Soit un état (ℓ, x) assigné à un nœud v de $\mathcal{G}_t$, par le run de l'automate A_Q, évaluant la requête Q sur t. La première composante ℓ, nous dit si le nœud v répond ($\ell \in \{s, \top'\}$) ou non ($\ell \in \{\eta, \top\}$) à la requête Q. La deuxième composante $x = 1$ (resp. $x = 0$), nous informe si le nœud v possède au moins un (resp. ne possède aucun) nœud descendant qui répond à Q. Par suite, conformément à la sémantique présentée dans la Table 4.1, les états $(s, 1)$ et $(\top', 1)$ seront appelés *sélectionnant*. Le concept d'utiliser des états en forme de couples est très pratique, car il permet de représenter la REPONSE à la requête Q, d'une façon concise. En effet, pour représenter la partie du rlag $\mathcal{G}_t$, qui contient toutes les réponses à Q, il suffit de garder seulement ces nœuds auxquels le run a assigné un état dont la deuxième composante est 1 (voir l'Exemple 4.4).

nom d'état	**assignable au nœud v tel que**
$(s,1)$	$llab(v) \neq \sigma$ et v est une réponse à Q
$(\eta,1)$	$llab(v) \neq \sigma$ et v n'est pas une réponse à Q, mais il existe un descendant de v répondant à Q
$(\eta,0)$	$llab(v) \neq \sigma$ et v n'est pas une réponse à Q, et aucun descendant de v n'est pas une réponse à Q
$(\top',1)$	$llab(v) = \sigma$ et v est une réponse à Q
$(\top,1)$	$llab(v) = \sigma$ et v n'est pas une réponse à Q, mais il existe un descendant de v répondant à Q
$(\top,0)$	$llab(v) = \sigma$ et v n'est pas une réponse à Q, et aucun descendant de v n'est pas une réponse à Q
init	$llab(v) = Root$ ou $llab(v) = Root_i$

Table 4.1. La sémantique des états de l'automate A_Q correspondant à $Q =$ `//*[axis::`σ`]`

Sur l'ensemble des ll–paires, on introduit un ordre partiel, dit *ordre de priorité* :

$$(\eta,0) > (\eta,1) > (s,1) \quad \text{et} \quad (\top,0) > (\top,1) > (\top',1).$$

L'unique run de A_Q évaluant la requête Q, sera appelé *run de priorité maximale*. Pour construire ce run, on utilise ce qu'on appelle une *stratégie de priorité maximale*, qui exige que le run r de l'automate A_Q sur le rlag $\mathcal{G}_t$ doive être une fonction totale, et pour tout nœud v de $\mathcal{G}_t$, la ll–paire $r(v)$ doive être la plus prioritaire possible parmi toutes les ll–paires qui peuvent être assignées à v en utilisant les transitions de A_Q. Cette stratégie explique le choix de priorité entre les ll–paires : on pose $(l,0) > (l,1)$ pour ne pas assigner $(l,1)$ à un nœud qui commence une branche ou il n'y a pas de nœuds répondant à Q, et $(l,1) > (l',1)$ pour ne pas sélectionner des nœuds qui ne répondent pas à Q. Voici la définition formelle du run de priorité maximale.

Définition 4.4. (MP) *Soient une requête de base Q, l'automate A_Q correspondant, un document t, et son rlag $\mathcal{G}_t$ associé. Une application $r\colon Nodes_{\mathcal{G}_t} \to States_Q$ est appelée run de priorité maximale de A_Q sur $\mathcal{G}_t$ si et seulement si, pour tout nœud v de $\mathcal{G}_t$, elle satisfait les conditions suivantes :*

- *$r(v)$ est défini (r est totale),*
- *$r(v)$ est déterminé par $llab(v)$,*
- *pour chaque nœud w parent de v, l'automate A_Q contient la transition de la forme $(r(w), llab(v)) \to r(v)$,*
- *$r(v)$ est la ll–paire maximale par rapport à l'ordre de priorité, qui satisfait les conditions précédentes.*

Remarquons que la deuxième et la troisième condition de la définition ci–dessus, impliquent que le run de priorité maximale de A_Q sur $\mathcal{G}_t$ est bien un run au sens de la définition 4.3. La quatrième condition de la Définition 4.4 garantit que le run de priorité maximale de A_Q sur $\mathcal{G}_t$ est unique. Grâce à cette unicité, on dira que l'automate A_Q est *non–ambigu*, et par la suite on ne sera intéressé que par le run de priorité maximale. Sans mention contraire, le mot run désignera le run

de priorité maximale. On prouve dans la Proposition 4.1, que le run de priorité maximale de A_Q sur $\mathcal{G}_t$, est le seul run qui permette d'évaluer la requête Q sur le document t. Dans la Section 4.5 on donne un algorithme qui construit le run de priorité maximale de l'automate A_Q, sur un rlag donné $\mathcal{G}_t$. On y montre également, qu'une telle construction est linéaire par rapport au nombre d'arêtes de $\mathcal{G}_t$.

4.3.1. Requêtes utilisant les axes verticaux

Dans cette section on présente les automates A_Q correspondant aux requêtes de base $Q =$ `//*[axis::`σ`]`, où `axis` est `self`, `child`, `parent`, `descendant`, `ancestor`.

Automate pour $Q =$ `//*[self::`σ`]`
La Figure 4.4 présente l'automate A_Q, permettant d'évaluer la requête de base $Q =$ `//*[self::`σ`]`. Cet automate n'a que quatre états : $init, (\eta, 0), (\eta, 1)$ et $(\top', 1)$.

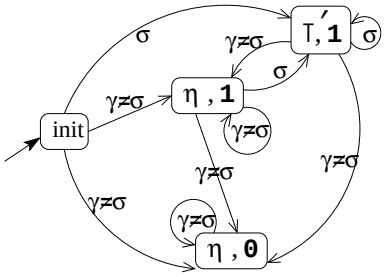


Figure 4.4. Automate A_Q, où $Q =$ `//*[self::`σ`]`

Ce sont les seules ll–paires conformes avec la sémantique donnée dans la Table 4.1. En effet, un nœud v d'un rlag donné $\mathcal{G}$ est sélectionné par Q si et seulement si $llab(v) = \sigma$. Autrement dit,

- tous les nœuds v de $\mathcal{G}$, tels que $llab(v) = \sigma$ constituent des réponses à Q (il n'existe pas de nœud auquel on pourrait assigner l'état $(\top, 0)$ ou $(\top, 1)$);
- aucun nœud v, tel que $llab(v) \neq \sigma$ n'est une réponse à Q (il n'existe pas de nœud auquel on pourrait assigner l'état $(s, 1)$).

Par conséquent, toutes les σ–transitions vont vers l'état sélectionnant $(\top', 1)$.
Il n'existe aucune transition partant d'état $(\eta, 0)$ (le seul état avec la deuxième composante 0), et allant vers $(\eta, 1)$ ou $(\top', 1)$ (les états avec la deuxième composante 1). Par suite, si le run de A_Q sur un rlag $\mathcal{G}$, assigne à un nœud v l'état $(\eta, 0)$, alors il assignera aussi l'état $(\eta, 0)$ à tous les nœuds u descendants de v. On illustre ce raisonnement dans l'exemple suivant :

Exemple 4.1. *La Figure 4.5 présente le run de priorité maximale de l'automate correspondant à la requête $Q =$* `//*[self::`b`]`*, sur le rlag $\mathcal{D}_t$ de la Figure 4.3. Remarquons que l'état le plus prioritaire possible, qui peut être assigné par le run de priorité maximale au nœud A_1, est $(\eta, 1)$. En effet, l'automate A_Q possède la transition $(init, \gamma \neq b) \rightarrow (\eta, 0)$, donc à priori on pourrait assigner à A_1 l'état $(\eta, 0)$ (qui est d'ailleurs plus prioritaire que $(\eta, 1)$). Mais si on le fait, on n'aura pas*

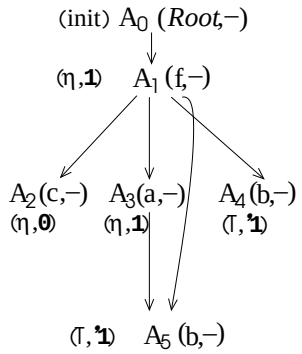


Figure 4.5. Évaluation de la requête de base $Q = //*[\texttt{self::}b]$

de transition applicable pour passer du nœud A_1 au nœud A_4 (dont le llab est b). Assigner $(\eta, 1)$ à A_1 est alors le seul moyen pour obtenir un run complet.

Automate pour $Q = //*[\texttt{child::}\sigma]$
Sur la Figure 4.6 on présente l'automate A_Q permettant d'évaluer la requête de base $Q = //*[\texttt{child::}\sigma]$. Les nœuds d'un rlag donné $\mathcal{G}$, qui sont sélectionnés par

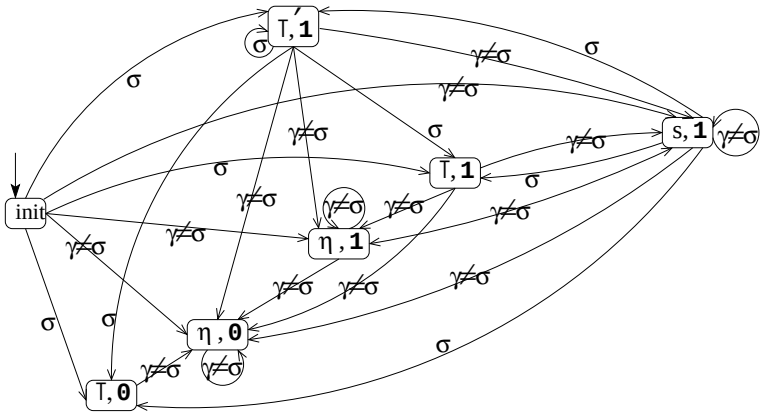


Figure 4.6. Automate A_Q, où $Q = //*[\texttt{child::}\sigma]$

Q, doivent avoir un enfant portant le llab σ. L'automate A_Q possède deux états sélectionnant : $(s, 1)$ et $(\top', 1)$. Soit v un nœud de $\mathcal{G}$, ayant un enfant portant le llab σ. Le run de A_Q sur $\mathcal{G}$ assigne à v : l'état $(s, 1)$ si $llab(v) \neq \sigma$, et l'état $(\top', 1)$ si $llab(v) = \sigma$. On explique maintenant l'absence de certaines transitions sur A_Q :

- Il n'existe aucune transition partant de $(\eta, 1)$ ou $(\top, 1)$, et allant vers $(\top, 0), (\top, 1)$ ou $(\top', 1)$. Cela est correct par rapport à la sémantique donnée dans la Table 4.1 : si le run de A_Q sur un rlag $\mathcal{G}$, assigne à un nœud u l'état $(\eta, 1)$ ou $(\top, 1)$, cela veut dire en particulier que u n'est pas une réponse à Q, il ne peut donc avoir aucun enfant portant le llab σ (rappelons que $(\top, 0), (\top, 1), (\top', 1)$ sont des états assignables aux nœuds avec le llab σ).

- La seule transition sortante des états du type $(\ell, 0)$ va vers $(\eta, 0)$. On sait que l'automate A_Q ne peut posséder aucune transition d'un état du type $(\ell, 0)$ vers un état du type $(\ell_1, 1)$. Voici pourquoi A_Q ne contient pas de transitions $((\ell, 0), \sigma) \to (\top, 0)$: en présence d'une telle transition il serait possible d'assigner à un nœud u la ll-paire $(\ell, 0)$ et à son enfant portant le llab σ la ll-paire $(\top, 0)$. Cela serait contradictoire avec la sémantique de la requête Q, car tout nœud u ayant un enfant portant le llab σ doit être sélectionné, et aucun état du type $(\ell, 0)$ n'est sélectionnant.

Automate pour $Q =$ `//*[parent::`σ`]`
L'automate pour `//*[parent::`σ`]` est présenté dans la Figure 4. . Les nœuds d'un

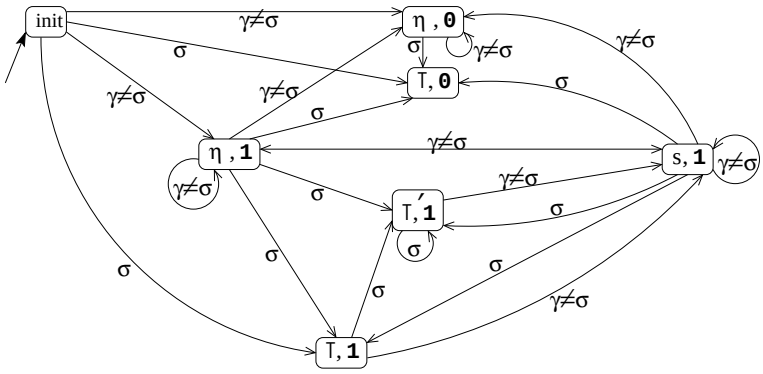


Figure 4.7. Automate A_Q, où $Q =$ `//*[parent::`σ`]`

rlag $\mathcal{G}$, répondant à cette requête, ont un parent portant le llab σ. C'est la raison pour laquelle toutes les transitions sortantes des états $(\top, 1)$ et $(\top', 1)$ (états assignables aux nœuds portant le llab σ) vont vers un des états sélectionnant $(\top', 1)$ ou $(s, 1)$.
Il n'y a aucune transition sortante d'état $(\top, 0)$, ce qui implique que cet état ne peut être assigné qu'aux feuilles (portant le llab σ) du rlag considéré. Cela est justifié par le fait, que si un nœud v de $\mathcal{G}$ porte le llab σ, et ce n'est pas une feuille, alors tous ces enfants doivent être sélectionnés. En particulier, v possède un descendant qui constitue une réponse à Q; d'où, le run de A_Q sur $\mathcal{G}$, ne peut pas assigner à v l'état $(\top, 0)$, car la sémantique de ce dernier interdit qu'il possède un descendant sélectionné.

Automate pour $Q =$ `//*[descendant::`σ`]`
La Figure 4.8 présente l'automate A_Q permettant d'évaluer la requête de base $Q =$ `//*[descendant::`σ`]`. Les nœuds répondant à Q sont ceux qui ont un descendant portant le llab σ.

- L'automate A_Q ne contient pas d'états du type $(l, 1)$. Leur absence est justifié par leur sémantique — ce sont des ll–paires assignables aux nœuds qui ne sont pas sélectionnés, mais qui ont un descendant sélectionné. Il suffit de remarquer que tout nœud v d'un rlag $\mathcal{G}$, qui a un descendant u sélectionné (c.–à–d., pour

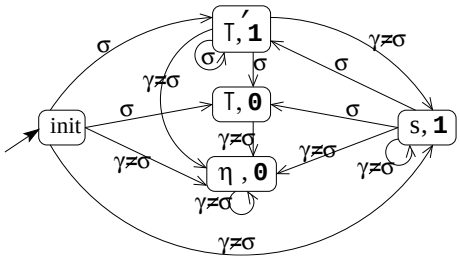


Figure 4.8. Automate A_Q, où $Q = //*[\texttt{descendant::}\sigma]$

lequel il existe un descendant w portant le llab σ) doit être lui même sélectionné, car w étant le σ–descendant de u, est aussi le σ–descendant de v.

- A_Q ne possède pas de transitions du type $((l,0),\sigma) \to (\top,0)$: si le run de A_Q sur $\mathcal{G}$ assigne à un nœud u l'état $(\top,0)$, cela implique en particulier que $llab(u) = \sigma$. Par suite, tout nœud v père de u, doit être sélectionné (car il possède un fils — donc descendant — portant le llab σ), et recevoir un état du type $(l',1)$ et non pas $(l,0)$).

Automate pour $Q = //*[\texttt{ancestor::}\sigma]$
Voici le dernier automate pour un axe vertical — Figure 4.9. Il correspond à la requête de base $Q = //*[\texttt{ancestor::}\sigma]$. La REPONSE à Q sur un rlag $\mathcal{G}$ est

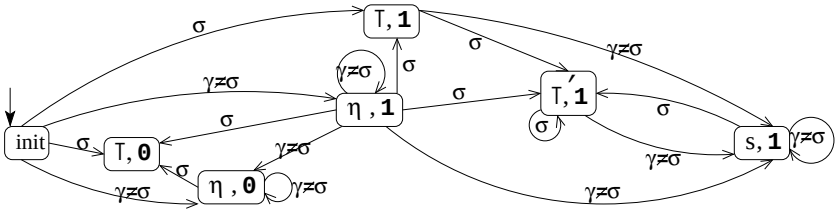


Figure 4.9. Automate A_Q, où $Q = //*[\texttt{ancestor::}\sigma]$

constituée de tous les nœuds de $\mathcal{G}$ qui ont un ancêtre portant le llab σ.

- Le top–down run de A_Q sur $\mathcal{G}$, assigne à un nœud v, qui n'est pas une feuille et qui porte le llab σ
  — l'état $(\top,1)$, si v n'a aucun nœud ancêtre portant le llab σ,
  — l'état $(\top',1)$, si v possède un nœud ancêtre portant le llab σ.
  Tous les nœuds u descendants de tels v doivent être sélectionnés — toutes les transitions sortantes de $(\top,1)$ ou $(\top',1)$ vont vers un des états sélectionnant $(s,1)$ ou $(\top',1)$.
- Il est évident que tout nœud descendant d'un nœud sélectionné doit être lui même sélectionné. D'où, toutes les transitions partant d'un état sélectionnant vont vers un état sélectionnant.
- Il n'existe aucune transition sortant de l'état $(\top,0)$. Par suite, cet état ne peut être assigné qu'aux nœuds feuilles portant le llab σ et n'ayant aucun ancêtre avec ce llab.

- Le lecteur peut être surpris par la présence des transitions du type

$$((\eta, 1), \delta) \to (l', 1),$$

pour $\delta \in \{\gamma, \sigma\}$. En effet, pourquoi sélectionner un nœud u (état $(l', 1)$) qui est l'enfant d'un nœud v dont le llab n'est pas σ, et dont aucun ancêtre ne porte ce llab (sémantique d'état $(\eta, 1)$)? Pourtant on a bien besoin de ces transitions, car n'oublions pas que l'on évalue les requêtes sur les documents qui peuvent être compressés. Grâce à une telle transition, on peut sélectionner un nœud u ayant plusieurs pères, parmi lesquels il y a au moins un (mais peut–être pas tous) qui porte le llab σ, ou a comme ancêtre un nœud portant ce llab.

4.3.2. Requêtes utilisant les axes horizontaux

Dans cette section on montre comment évaluer les requêtes de base utilisant les axes du type `sibling`. On a déjà mentionné que les automates correspondant à ces requêtes courent sur l'ensemble des chaînons et non pas directement sur le rlag de dépendance. Comme les chaînons sont des graphes linéaires, les automates pour les axes horizontaux sont plus simples que ceux pour les axes verticaux, car ils n'ont pas à tenir compte des nœuds ayant plusieurs pères. On commence par présenter les deux automates en question. Ensuite, on explique comment, à partir des évaluations sur les chaînons, obtenir la REPONSE à la requête donnée sur le rlag de dépendance.

Automate pour $Q =$ `//*[following-sibling::`σ`]`
La Figure 4.10 présente l'automate A_Q, pour l'évaluation de la requête de base $Q =$ `//*[following-sibling::`σ`]`. Considérons un document (trdag) t, et l'ensemble

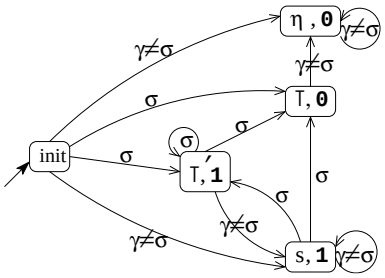
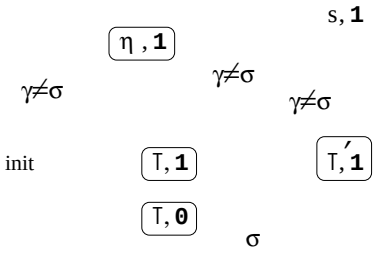


Figure 4.10. Automate A_Q, où $Q =$ `//*[following-sibling::`σ`]`

des chaînons $\mathcal{F}$ de la grammaire $\mathcal{L}_t$ correspondante. Les nœuds de t répondant à cette requête doivent avoir un frère suivant qui porte le llab σ. Remarquons que la notion de frères suivants sur le trdag t correspond à celle de descendants sur les chaînons. Par suite, pour tout chaînon $\mathcal{G}$ de $\mathcal{L}_t$, l'automate A_Q sélectionne tout nœud de $\mathcal{G}$, ayant un descendant avec le llab σ. D'où, l'automate A_Q est une version simplifiée de l'automate pour `//*[descendant::`σ`]` — simplifiée, car il court sur des graphes linéaires, où chaque nœud a un seul père. (Voir le paragraphe sur l'automate évaluant la requête `//*[descendant::`σ`]` pour la sémantique des transitions de A_Q.)

Automate pour $Q = //*[\texttt{preceding-sibling::}\sigma]$
La dernière requête de base à considérer est $Q = //*[\texttt{preceding-sibling::}\sigma]$. La Figure 4.11 présente l'automate A_Q évaluant Q. La notion de frères précédents sur



Ensuite, à chaque nœud v de $\mathcal{D}_t$, on déduit un nouveau label, en utilisant la fonction suivante :

$$\lambda_{\widehat{R}}(v) = \begin{cases} (s,-), & \text{si } ll_{\widehat{R}}(v) \cap \{(s,1),(\top',1)\} \neq \emptyset, \\ (\eta,-), & \text{si } ll_{\widehat{R}}(v) \cap \{(s,1),(\top',1)\} = \emptyset. \end{cases}$$

Finalement, en faisant courir l'automate correspondant à la requête `//*[self::`s`]`, sur le rlag $\mathcal{D}_t$ ainsi relabelé (noté $\lambda_{\widehat{R}}(\mathcal{D}_t)$), on obtient le run

$$\widehat{r}\colon Nodes_{\lambda_{\widehat{R}}(\mathcal{D}_t)} \to \{init, (\top',1), (\eta,1), (\eta,0)\}, \qquad (\widehat{r})$$

qui nous donne la REPONSE à notre requête Q, sur le rlag $\mathcal{D}_t$:

- $(\top',1)$ pour un nœud sélectionné,
- $(\eta,1)$ pour un nœud qui n'est pas sélectionné, mais qui à un descendant sur $\mathcal{D}_t$ qui l'est,
- $(\eta,0)$ pour un nœud qui n'est ni sélectionné, ni n'a aucun descendant sélectionné.

Exemple 4.2. *On illustre l'évaluation d'une requête contenant un axe horizontal sur la Figure 4.12. On y considère la requête* $Q =$`//*[following-sibling::`b`]`*, et le document t de la Figure 4.3.*

4.4. Résultats sur le run de priorité maximale

Dans cette section on considère une requête de base $Q =$ `//*[axis::`σ`]`, et l'automate A_Q correspondant. On fixe également un trdag t. Rappelons qu'on note par $\mathcal{G}$ le rlag $\mathcal{D}_t$ si `axis` est un axe vertical, et un des chaînons de $\mathcal{L}_t$ si `axis` est un axe horizontal. L'objectif de cette section est de montrer que le run de priorité maximale **MP** (Définition 4.4, page 48) est le seul run de A_Q sur $\mathcal{G}$, qui permette d'évaluer la requête Q sur le document t.

Définition 4.5. *Une fonction* $\mathbb{L}\colon Nodes_{\mathcal{G}} \to States_Q$ *est dite conforme avec la sémantique de la requête Q, ssi elle assigne, à chaque nœud de $\mathcal{G}$, un état conforme avec la sémantique représentée dans la Table 4.1.*

On commence par un théorème qui dit que tout run complet de A_Q sur $\mathcal{G}$, qui est conforme avec la sémantique de Q, satisfait la condition de priorité maximale **MP**.

Théorème 4.1. *Soit une fonction* $\mathbb{L}\colon Nodes_{\mathcal{G}} \to States_Q$*, qui est conforme avec la sémantique de la requête Q. Il existe un run complet r de l'automate A_Q sur $\mathcal{G}$, tel que :*

(i) r soit compatible avec $\mathbb{L}$, c.-à-d., $r(v) = \mathbb{L}(v)$ pour tout nœud v de $\mathcal{G}$,
(ii) r vérifie la condition de priorité maximale **MP** *(Définition 4.4).*

Preuve. On construit d'abord un run complet satisfaisant la condition (i). Cette construction est spécifique pour chaque requête de base Q, car elle dépend de l'automate A_Q correspondant. Pour cette construction, on utilise la récurrence sur l'ensemble des nœuds de $\mathcal{G}$. On présente les détails dans le cas de $Q =$ `//*[parent::`σ`]`. Pour les autres requêtes de base le raisonnement est similaire.

Puisque l'axe de navigation de la requête considérée est `parent`, le graphe $\mathcal{G}$ est le rlag $\mathcal{D}_t$. Soit v la racine de $\mathcal{D}_t$. On pose $r(v) = \mathbb{L}(v) = init$. Soit u l'enfant de

r_1 on F_1 :

$A_1^0(Root_1,-)$ (init)

$\downarrow$

$A_2(c,-)$ (s,**1**)

$A_3(a,-)$ (s,**1**)

$A_4(b,-)$ (⊤',**1**)

$A_5(b,-)$ (⊤,**0**)

$A_2(c,-)$ (η,**0**)

r_0 on F_0 :

$A_0^0(Root_0,-)$ (init)

$\downarrow$

$A_1(f,-)$ (η,**0**)

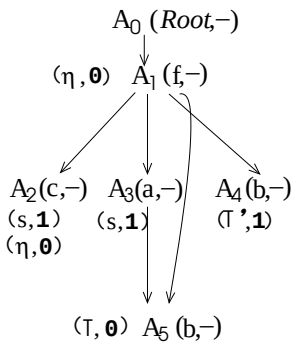
r_3 on F_3 :

$A_3^0(Root_3,-)$ (init)

$\downarrow$

$A_5(b,-)$ (⊤,**0**)

r_0, r_1, r_3 on D_t :



run de l'automare posr.06 l 4286.52 6735.93 9((

$\frown(D_t)$

(

$\mathbb{L}(w)$	$\mathbb{L}(u)$
$(\top, 0)$	$(\eta, 0)$
$(\top, 0)$	$(\top, 0)$
$(\top, 1)$	$(\top, 0)$
$(\top, 1)$	$(\eta, 0)$
$(\top, 1)$	$(\top, 1)$

$\mathbb{L}(w)$	$\mathbb{L}(u)$
$(\top, 1)$	$(\eta, 1)$
$(\top', 1)$	$(\top, 0)$
$(\top', 1)$	$(\top, 1)$
$(\top', 1)$	$(\eta, 0)$
$(\top', 1)$	$(\eta, 1)$

Puisque la fonction $\mathbb{L}$ est conforme avec la sémantique de la requête `//*[parent::`σ`]`, et les seules possibilités pour $\mathbb{L}(w)$ sont $(\top, 0), (\top, 1)$ ou $(\top', 1)$, on peut en déduire que $llab(w) = \sigma$. Le nœud u possède alors un parent σ, et il doit donc être sélectionné par la requête Q. D'où aucune des possibilités pour $\mathbb{L}(u)$ du tableau ci–dessus n'est conforme avec la sémantique de Q, ce qui est contradictoire avec les hypothèses du théorème.

(*ii*) Il suffit maintenant de démontrer que le run ainsi construit satisfait la condition de priorité maximale. Cette partie de la preuve ne dépend pas de la requête de base considérée. Écrivons alors Q sous une forme générale $Q =$ `//*[axis::`σ`]`, pour un axe de base `axis` quelconque, et un certain σ donné. Supposons que le run r construit plus haut ne satisfait pas **MP**. Il existe alors un nœud v sur $\mathcal{G}$, tel que $r(v) = \mathbb{L}(v) = (l, 1)$, et la ll–paire la plus prioritaire compatible avec $r(w)$, pour tout les nœuds w parents de v, est $(l, 0)$ (ou respectivement $r(v) = (l', 1)$ et la ll–paire la plus prioritaire est $(l, 1)$ ou $(l, 0)$). Considérons le premier cas. Si la ll–paire la plus prioritaire pour $r(v)$ est $(l, 0)$, alors suivant la sémantique, le nœud v ne possède aucun descendant sélectionné par Q, ce qui dirait que la fonction $\mathbb{L}$ n'est pas conforme avec la sémantique de la requête Q. Contradiction. Le raisonnement dans l'autre cas est similaire.

□

Montrons maintenant la réciproque du Théorème 4.1. Le Théorème 4.2 dit que tout run r, satisfaisant la condition **MP**, est conforme avec la sémantique de Q.

Théorème 4.2. *Soit r un run complet de l'automate A_Q sur le rlag $\mathcal{G}$, qui satisfait la condition de priorité maximale* **MP**. *La fonction $\mathbb{L}$ allant de $Nodes_{\mathcal{G}}$ vers l'ensemble de ll–paires, et définie pour tous les nœuds v de $\mathcal{G}$ par $\mathbb{L}(v) = r(v)$, est conforme avec la sémantique de la requête Q.*

Preuve. On raisonne par absurde. Supposons que la fonction $\mathbb{L}$, définie par $\mathbb{L}(v) = r(v)$, n'est pas conforme avec la sémantique de la requête Q. Le raisonnement est spécifique pour chaque requête de base Q, car il dépend des transitions de A_Q correspondant. On donne ici la preuve dans le cas $Q =$ `//*[descendant::`σ`]`. Les preuves dans les autres cas sont analogues.

L'axe `descendant` est un axe vertical, donc le graphe $\mathcal{G}$ désigne le rlag $\mathcal{D}_t$. L'automate considéré A_Q n'a que cinq états : *init*, $(\eta, 0)$, $(s, 1)$, $(\top, 0)$, $(\top', 1)$. L'hypothèse par absurde implique qu'il existe un nœud u sur $\mathcal{D}_t$, tel que : pour tous les nœuds v ancêtres de u, la valeur $\mathbb{L}(v) = r(v)$ est conforme avec la sémantique de la requête Q, mais la valeur $r(u)$ ne l'est pas. On a donc les possibilités suivantes :

cas	$r(u)$ actuel	$llab(u)$	caractéristique de u	$r(u)$ correct
a	$(\top', 1)$	σ	u n'est pas une réponse à Q	$(\top, 0)$
b	$(s, 1)$	$\gamma \neq \sigma$	u n'est pas une réponse à Q	$(\eta, 0)$
c	$(\eta, 0)$	$\gamma \neq \sigma$	u est une réponse à Q	$(s, 1)$
d	$(\top, 0)$	σ	u est une réponse à Q	$(\top', 1)$

Dans tous ces quatre cas, on doit montrer que :

1. l'état donné dans la dernière colonne du tableau peut être atteint à partir de toutes les ll–paires assignées aux nœuds pères de u,
2. en remplaçant la valeur $r(u)$ donnée dans la deuxième colonne, par la valeur $r(u)$ donnée dans la dernière colonne, la fonction
   $\mathcal{D}_t$ tout entier.

Le cas (a):
On sait que le llab du nœud u est σ, et que u ne possède aucun descendant σ. En particulier, aucun des nœuds au–dessous de u ne peut être sélectionné par la requête $Q =$ //$*$ σ]. D'où, la seule possibilité pour correctement assigner un état à un nœud descendant de u, c'est . Les propriétés demandées 2 sont alors vérifiées, par les observations suivantes :

1. pour tout état (ℓ, x) de l'automate A_Q, si A_Q possède la transition $((\ell, x), \sigma) \rightarrow (\top', 1)$, alors il possède aussi la transition $((\ell, x), \sigma) \rightarrow (\top, 0)$,
2. l'état $(\eta, 0)$, qui est le seul état pouvant être assigné aux enfants de $(\top, 0)$, et à partir de lui même.

Le raisonnement dans le cas b est analogue.

Les cas (c) et (d):
En réalité, on va montrer que les cas c et d ne peuvent exister. On sait que le nœud u constitue une réponse à la requête $Q =$ //$*$[descendant::σ]. Il existe donc un nœud w descendant de u, tel que $llab(w) = \sigma$. Les seuls états qui peuvent être alors assignés par un run de A_Q, à ce nœud w, sont $(\top, 0)$ ou $(\top', 1)$. Cela veut dire, que sur le chemin entre le nœud u et w, le run r $(l, 0)$ vers $(\top, 0)$ ou $(\top', 1)$, ce qui est impossible par simple observation des transitions de l'automate. Contradiction.

□

Les deux théorèmes précédents nous donnent le résultat suivant :

Proposition 4.1. *Le run complet r de l'automate A_Q sur un trdag donné $\mathcal{G}$, est conforme avec la sémantique de la requête*

A_Q sur $\mathcal{G}$, qui permet d'évaluer la requête Q sur .

4.5. Algorithme pour le run de priorité maximale

Soient une requête de base $Q =$ //$*$[axis::σ], l'automate correspondant A_Q, un trdag t, et le rlag associé (conformément à axis) . Dans cette section, on présente

un algorithme qui permet de construire le run de priorité maximale de l'automate A_Q sur $\mathcal{G}$. Notons que, si `axis` est un axe horizontal, alors le graphe $\mathcal{G}$ représente un chaînon, et la construction du run de priorité maximale de A_Q sur $\mathcal{G}$ est triviale, car chaque chaînon est un graphe linéaire. Sans perte de généralité, on peut donc supposer que `axis` est un axe vertical, d'où $\mathcal{G}$ représente le rlag $\mathcal{D}_t$.

Soit n (resp. $n+1$) le nombre de nœuds sur t (resp. sur $\mathcal{D}_t$). L'idée est de construire un dag $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ qui représenterait tous les runs complets de A_Q sur $\mathcal{D}_t$, et ensuite en choisir celui qui est de priorité maximale. Les sommets de $\mathbb{G}$ seront de la forme :

- $A_{i,init}$, pour $i = 0$,
- et $A_{i,\alpha}$, pour $i \in \{1, \ldots, n\}$,

où A_i sera un non–terminal de $\mathcal{L}_t$, et α une ll–paire appartenant à $States_Q$, déterminée par le llab de nœud sur $\mathcal{D}_t$, portant le nom A_i. Par abus de notation, on dénotera par $llab(A_i)$, le llab du nœud v sur $\mathcal{D}_t$, tel que $name(v) = A_i$.

Le coût total de l'algorithme est $\mathcal{O}(m)$, où $m = |\mathcal{D}_t|$ est le nombre d'arêtes du rlag $\mathcal{D}_t$. En particulier, si t est un arbre, son rlag $\mathcal{D}_t$ est isomorphe à t, alors l'algorithme est linéaire par rapport au nombre de nœuds de t. D'où, la complexité de notre approche est comparable avec celle des autres approches d'évaluations connues ([16, 35, 41, 42]).

L'algorithme contient quatre pas, que nous décrivons maintenant :

1. La forme initiale du graphe $\mathbb{G}$:
   (*i*03980 36191990 Ω ( )500Ω(Ω3696020 Ω(41836020 Ω(:) 96119020 Ω() Ω6960160 Ω)

pour ouet (*A* 2ΩΩ39108(≠)322266]Ω329109091 Ω2(7)340 Ω[( )350[( )476908( )356804

déterminée par $llab(A_j)$, et satisfaisant pour tout $i < j$ la condition suivante : si le rlag $\mathcal{D}_t$ possède l'arête (A_i, A_j), alors sur $\mathbb{G}$ il existe une arête $(A_{i,r(A_i)}, A_{j,\beta})$ qui est marqué correcte (symbole booléen 1).

INPUT :
$\{A_i, 1 \leq i \leq n\}$ — l'ensemble des non–terminaux de $\mathcal{L}_t$,
E — l'ensemble des arêtes de $\mathcal{D}_t$,
n — le nombre de nœuds sur t,
$m = |\mathcal{D}_t|$ — le nombre d'arêtes sur $\mathcal{D}_t$,
Δ_Q — l'ensemble des transitions de l'automate A_Q,
$States_Q$ — l'ensemble des états de l'automate A_Q,
$States(llab(A_i))$ — l'ensemble des (≤ 3) états, détrminés par $llab(A_i)$.

BEGIN:
/* 1(i): Potential edges on $\mathbb{G}$ */
For all $(A_i, A_j) \in E$ do
For all $\alpha \in States(llab(A_i))$ and $\beta \in States(llab(A_j))$ do
set $(A_{i,\alpha}, A_{j,\beta}) := 0$;
/* 1(ii): Edges compatible with Δ_Q */
For all $(A_i, A_j) \in E$ do
For all $\alpha \in States(llab(A_i))$ and $\beta \in States(llab(A_j))$ do
If $(\alpha, llab(A_j)) \to \beta \in \Delta_Q$ then set $(A_{i,\alpha}, A_{j,\beta}) := 1$;
/* 2: Top–Down pruning */
For j from 0 to n do
For all $\beta \in States(llab(A_j))$ do
If **(i)** exists $(A_i, A_j) \in E$ such that
for all $\alpha \in States(llab(A_i))$ we have $(A_{i,\alpha}, A_{j,\beta}) = 0$,
or **(ii)** exists $(A_j, A_k) \in E$ such that
for all $\gamma \in States(llab(A_k))$ we have $(A_{j,\beta}, A_{k,\gamma}) = 0$
then { set $(-, A_{j,\beta}) := 0$; set $(A_{j,\beta}, -) := 0$;
/** Here $(-, A_{j,\beta})$ and $(A_{j,\beta}, -)$ respectively stand
for *any* incoming and outgoing edge at $A_{j,\beta}$ **/
}
/* 3: Bottom–Up pruning */
For j from n downto 0 do
For all $\beta \in States(llab(A_j))$ do
If **(i)** exists $(A_i, A_j) \in E$ such that
for all $\alpha \in States(llab(A_i))$ we have $(A_{i,\alpha}, A_{j,\beta}) = 0$,
or **(ii)** exists $(A_j, A_k) \in E$ such that
for all $\gamma \in States(llab(A_k))$ we have $(A_{j,\beta}, A_{k,\gamma}) = 0$
then { set $(-, A_{j,\beta}) := 0$; set $(A_{j,\beta}, -) := 0$; }
/* 4: Constructing the maximal priority run r */
$r(0) = init$; $j := 1$;
While $j \leq n$ do
{
$\delta := state$ of maximal priority in $States(llab(A_j))$;

(#) If for all $(A_i, A_j) \in E$ we have $(A_{i,r(i)}, A_{j,\delta}) = 1$
then set $r(j) := \delta$;
else
{ $\delta --$; GOTO (#);
/** $\delta --$: next >–maximal state in $States(llab(A_j))$ **/
}
$j := j + 1$;
}
END.

Cet algorithme calcule, en temps $\mathcal{O}(|\mathcal{D}_t|)$, le run de priorité maximale de l'automate A_Q sur le trdag $\mathcal{D}_t$. En effet, la construction dans le pas 1 est bien dans $\mathcal{O}(|\mathcal{D}_t|) = \mathcal{O}(m)$, quant aux autres pas, il suffit de noter que :

$$\sum_{j=1}^{n} \#Parents(A_j) = \sum_{j=1}^{n} \#\{(A_i, A_j) \in E\} = m =$$

$$\sum_{i=1}^{n} \#Sons(A_i) = \sum_{i=1}^{n} \#\{(A_i, A_j) \in E\}.$$

4.6. Évaluation de requêtes composées

Nous montrons maintenant comment évaluer les requêtes qui ne sont pas basiques. On les appellera requêtes *composées*. Ce sont des requêtes *conjonctives* ou *disjonctives*, du type `//*[axis::`σ *conn* `axis'::`σ'`]`, où $conn \in \{and, or\}$, ou *imbriquées* `//*[axis`$_1$`::*[axis`$_2$`::*[... [axis`$_{k-1}$`::*[`S_k`]]...]]]`, où $k > 1$, et S_k est soit de la forme `axis::`σ, soit (`axis::`σ *conn* `axis'::`σ').

On commence par des requêtes conjonctives ou disjonctives. Soient une requête $Q =$ `//*[axis`$_1$`::`σ_1 *conn* `axis`$_2$`::`σ_2`]`, où $conn \in \{and, or\}$, et t un trdag donné. Remarquons que la REPONSE à Q sur t est tout simplement l'union, si $conn = or$, et l'intersection, si $conn = and$, de la REPONSE à $Q_1 =$`//*[axis`$_1$`::`σ_1`]` sur t, avec la REPONSE à $Q_2 =$`//*[axis`$_2$`::`σ_2`]` sur t. D'où, pour évaluer Q sur t, on fera courir les automates A_{Q_1} et A_{Q_2} correspondant respectivement aux Q_1 et Q_2, sur le rlag $\mathcal{D}_t$, ou sur l'ensemble des chaînons $\mathcal{F}$ de $\mathcal{L}_t$ (suivant les types des axes `axis`$_1$ et `axis`$_2$). On obtient ainsi deux fonctions r_i, pour $i \in \{1, 2\}$, allant de $Nodes_{\mathcal{D}_t}$ vers un ensemble de ll–paires;

- r_i étant le run de priorité maximale de A_{Q_i} sur $\mathcal{D}_t$, si `axis`$_i$ est un axe vertical,
- r_i étant la fonction $\widehat{r}$ définie à la page 55, si `axis`$_i$ est un axe horizontal.

La REPONSE à Q sur t est ensuite déduite par l'intermédiaire de la fonction

$$AND\colon Nodes_{\mathcal{D}_t} \to \{init, (\eta, 0), (\eta, 1), (s, 1)\},$$

si $conn = and$:

$$AND(v) = \begin{cases} init, & \text{si } r_1(v) = init, \text{ et } r_2(v) = init, \\ (s,1), & \text{si } r_1(v) = (l',1), \text{ et } r_2(v) = (l',1), \\ (\eta,0), & \text{si } r_1(v) = (l,0), \text{ ou } r_2(v) = (l,0), \\ (\eta,1), & \text{ailleurs;} \end{cases}$$

ou la fonction OR: $Nodes_{\mathcal{D}_t} \rightarrow \{init, (\eta,0), (\eta,1), (s,1)\}$, si $conn = or$:

$$OR(v) = \begin{cases} init, & \text{si } r_1(v) = init, \text{ et } r_2(v) = init, \\ (s,1), & \text{si } r_1(v) = (l',1), \text{ ou } r_2(v) = (l',1), \\ (\eta,0), & \text{si } r_1(v) = (l,0), \text{ et } r_2(v) = (l,0), \\ (\eta,1), & \text{ailleurs.} \end{cases}$$

La sémantique de ces deux fonctions est la suivante : les nœuds de $\mathcal{D}_t$, auxquels ces fonctions associent l'état $(s,1)$, répondent à la requête Q et sont donc sélectionnés; ceux qui ont reçu l'état $(\eta,1)$ ne sont pas sélectionnés mais se trouvent sur une branche qui mène vers un nœud répondant à Q; et ceux qui ont reçu $(\eta,0)$ ne sont pas sélectionnés, et n'ont aucun descendant qui le soit.

Exemple 4.3. *Sur la Figure 4.13, on illustre l'évaluation de la requête conjonctive* $Q =$ `//*[self::`b *and* `parent::`a`]`*, sur le trdag* t *de la Figure 4.3.*

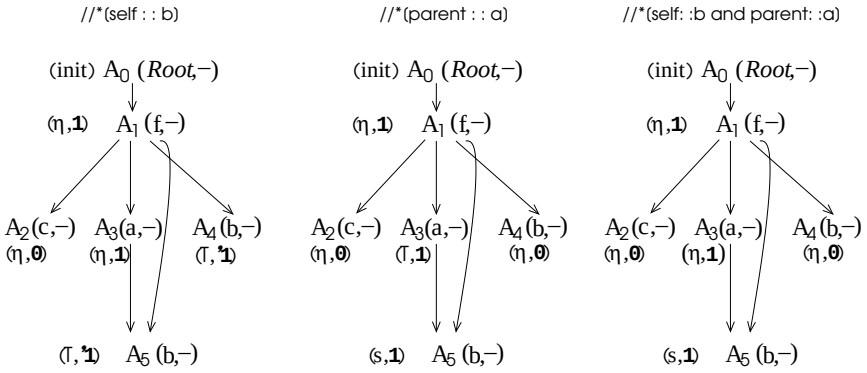


Figure 4.13. Évaluation de la requête conjonctive $Q =$ `//*[self::`b *and* `parent::`a`]`

Considérons maintenant le cas d'une requête imbriquée, qui est de la forme

$$Q = \texttt{//*[axis}_1\texttt{::*[}\ldots\texttt{[axis}_{k-1}\texttt{::*[}S_k\texttt{]]}\ldots\texttt{]]},$$

où $k > 1$, $conn \in \{and, or\}$, et S_k est de la forme :

- $S_k =$ `axis`$_k$`::`σ_k, ou
- $S_k =$(`axis`$_k$`::`σ_k *conn* `axis'`$_k$`::`σ'_k).

Pour $i \in \{1, \ldots, k-1\}$, notons par S_i l'expression `axis`$_i$`::*`, et posons $\sigma_i = *$. Soit t un trdag donné. Pour évaluer Q sur t, on procédera d'une façon récursive décroissante sur $i \in \{k, \ldots, 1\}$. Posons $r_{k+1}(\mathcal{D}_t) = \mathcal{D}_t$. On trouve d'abord la REPONSE

correspondante à la partie $[S_k]$ sur le rlag $\mathcal{D}_t$ (en évaluant la requête $//*[S_k]$). Ensuite, on relabelle le rlag $\mathcal{D}_t$, en utilisant la fonction de relabeling $label_k$, dont la définition est présentée plus bas. On obtient ainsi le rlag relabelé, qui sera noté $r_k(\mathcal{D}_t)$. Ensuite, on continue l'évaluation de Q, en faisant courir sur ce rlag relabelé l'automate correspondant à la requête de base $//*[\texttt{axis}_{k-1}::s]$. On procède ainsi, en relabelant à chaque fois le rlag $r_{i+1}(\mathcal{D}_t)$ à l'aide de la fonction $label_i$, et en courant avec l'automate correspondant à $//*[\texttt{axis}_{i-1}::s]$, sur ce rlag relabelé appelé $r_i(\mathcal{D}_t)$. A la fin d'une telle évaluation, on obtient le rlag $r_1(\mathcal{D}_t)$, qui représente la REPONSE à la requête Q sur t. Soit $i \in \{1, \ldots, k\}$. Voici la définition de la fonction de relabeling $label_i$, dont le rôle est de relabeler tous les nœuds du trdag $\mathcal{D}_t$, après avoir évalué la partie $[S_i]$ et avant l'évaluation de la partie $[S_{i-1}]$:

$$label_i \colon Nodes_{r_{i+1}(\mathcal{D}_t)} \rightarrow \{(Root, -)(s,1),(\eta,1),(\eta,0)\}$$

$$label_i(v) = \begin{cases} (Root,-) & \text{si } r_i(v) = init, \\ (s,1), & \text{si } r_i(v) \in \{(s,1),(\top',1)\}, \\ (\eta,1), & \text{si } r_i(v) \in \{(\eta,1),(\top,1)\}, \\ (\eta,0), & \text{si } r_i(v) \in \{(\eta,0),(\top,0)\}. \end{cases}$$

On explique maintenant ce que représente la fonction r_i utilisée dans la définition de la fonction de relabeling $label_i$. On note par $r_i(\mathcal{D}_t)$, le rlag $r_{i+1}(\mathcal{D}_t)$ relabelé par la fonction $label_i$. On a :

- si $S_i = \texttt{axis}_i::\sigma_i$, et $\texttt{axis}_i$ est un axe vertical, alors la fonction r_i représente le run de priorité maximale de l'automate correspondant à la requête de base $Q_i = //*[\texttt{axis}_i::\sigma_i]$ sur le rlag $r_{i+1}(\mathcal{D}_t)$;
- si $S_i = \texttt{axis}_i::\sigma_i$, et $\texttt{axis}_i$ est un axe horizontal, alors la fonction r_i représente la fonction $\widehat{r}$ correspondant à la requête $Q_i = //*[\texttt{axis}_i::\sigma_i]$, définie à la page 55, Section 4.3.2;
- si $S_k = (\texttt{axis}_k::\sigma_k \ \ conn \ \ \texttt{axis'}_k::\sigma'_k)$, alors la fonction r_k représente la fonction AND ou OR conformément à $conn$.

La REPONSE à notre requête imbriquée Q est représentée par le rlag relabelé $r_1(\mathcal{D}_t)$. Elle est composée de tous les nœuds v de ce rlag qui portent le label $label_1(v) = (s,1)$. On illustre l'évaluation d'une requête composée imbriquée dans l'exemple ci–dessous.

Exemple 4.4. *On évalue la requête* $/\texttt{descendant::}*[\texttt{descendant::}b[\texttt{parent::}a]]$, *sur le document partiellement compressé t, représenté sur la Figure 4.3, à la page 44. Pour pouvoir utiliser notre approche, on transforme la requête donnée, en une requête Q standardisée équivalente :*

$$Q = //*\,[\underbrace{\texttt{descendant::}*}_{S_1}\,[\underbrace{\texttt{self::}b \ \ and \ \ \texttt{parent::}a}_{S_2}]].$$

L'évaluation de Q commence par la recherche de la REPONSE correspondante à la partie $S_2 =$ $\texttt{self::}b \ and \ \texttt{parent::}a$. *On trouve cette REPONSE, en évaluant la requête* $//*[\texttt{self::}b \ and \ \texttt{parent::}a]$ *sur le rlag $\mathcal{D}_t$ (voir la Figure 4.13). Ensuite, on relabelle le rlag représenté à droite de la Figure 4.13, conformément à la fonction de relabeling $label_2$. On obtient le rlag appelé $r_2(\mathcal{D}_t)$, qui est représenté à gauche de*

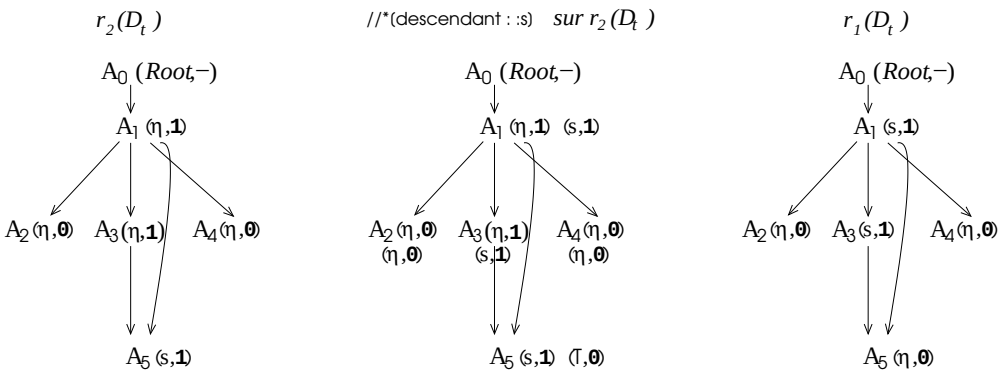


Figure 4.14. Évaluation de la requête imbriquée Q = //∗[descendant::∗[self::b *and* parent::a]]

la Figure 4.14. Ensuite, sur ce rlag $r_2(\mathcal{D}_t)$*, on fait courir l'automate correspondant à la requête* //∗[descendant::s] *(Figure 4.14 au milieu). Finalement, on relabelle le rlag* $r_2(\mathcal{D}_t)$*, conformément à la fonction de relabeling* $label_1$*. On obtient ainsi le rlag* $r_1(\mathcal{D}_t)$ *(la Figure 4.14 à droite), qui représente la REPONSE (nœuds labelés par* $(s, 1)$*) à* Q *sur* t*. Remarquons, que toutes les réponses à* Q *sur* $\mathcal{D}_t$*, se trouvent dans le sous–trdag de* $r_1(\mathcal{D}_t)$*, composé par les nœuds ayant des labels avec la composante booléenne* 1*. Pour garder le sous–trdag de* $\mathcal{D}_t$ *qui contient toutes les réponses à* Q*, il suffit de couper tous les nœuds ayant des labels avec la composante booléenne* 0 *et des arêtes menant vers ces nœuds (en pointillé sur la Figure 4.15).*

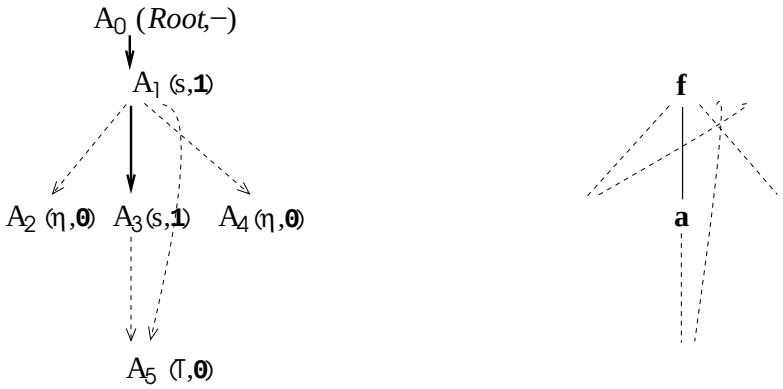


Figure 4.15. Sous-trdags de $\mathcal{D}_t$ et t, contenant toutes les réponses à Q

Pour finir cette section, notons que la complexité d'évaluation d'une requête composée Q sur un graphe $\mathcal{G}$, en utilisant l'approche présentée dans ce chapitre, est $\mathcal{O}(|Q||\mathcal{G}|)$. En effet, comme on a vu dans la Section 4.5, l'évaluation d'une requête de base sur $\mathcal{G}$ est dans $\mathcal{O}(|\mathcal{G}|)$, et $|Q|$ est le nombre de requêtes de base qui composent Q. La complexité de notre approche est donc la même que la meilleure complexité

connue pour le problème d'évaluation des requêtes XPath sur les documents XML compressés, modélisés à l'aide des dags (voir [16, 35]).

4.7. Réponse à une requête sur arbre équivalent

Soient une requête Core XPath Q, et t un trdag donné. Rappelons que la REPONSE à Q sur t est un ensemble noté $R_Q^t \subseteq Nodes_t$. L'objectif de cette section est de montrer comment, à partir de l'évaluation de Q sur t, déduire la REPONSE $R_Q^{\widehat{t}}$ à la même requête Q sur l'arbre $\widehat{t}$ équivalent de t. Cette question est d'une grande importance, car le modèle classique d'un document XML est arborescent, et comme on peut le voir dans l'exemple ci–dessous, la REPONSE à Q sur t ne correspond pas toujours à la REPONSE à Q sur $\widehat{t}$.

Exemple 4.5. *Considérons la requête de base* $Q =$ `//*[parent::b]`. *Soit le trdag* t *présenté à gauche de la Figure 4.16, et son arbre équivalent* $\widehat{t}$ *présenté à droite. Il y a deux nœuds qui répondent à* Q *sur* t *(celui qui représente les positions* $1, 21, 3$*, et celui représentant les positions* $111, 2111, 311$*), car chacun d'eux a un père 'b'. Pourtant, si on évalue* Q *sur l'arbre* $\widehat{t}$*, on obtient la REPONSE* $111, 21, 2111, 311$*, et les nœuds aux positions* 1 *et* 3 *ne répondent pas à* Q *sur* $\widehat{t}$*.*

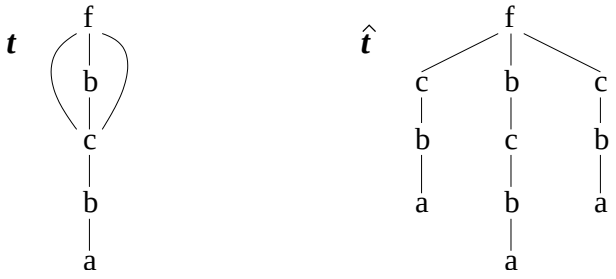


Figure 4.16. trdag t et son arbre équivalent $\widehat{t}$

La situation décrite dans l'Exemple 4.5, est due au fait qu'il y a plusieurs chemins qui lient la racine de t avec le nœud portant le nom c : le chemin correspondant à la position 1, celui correspondant à la position 21, et finalement celui correspondant à la position 3. Le nœud portant le nom c est sélectionné sur t à cause de la position 21 (c'est sur le chemin correspondant qu'on trouve le père b), et non pas à cause des positions 1 ni 3. Le but de l'approche présentée dans cette section est d'affiner la notion du run de nos automates, pour qu'il tienne aussi compte du chemin parcouru vers un nœud (ou plutôt une position) sélectionné(e) sur un trdag. Notons que dans le cas où t est un arbre, la REPONSE à une requête Q restera inchangée, car chaque nœud d'un arbre représente une et une seule position. Notons aussi, que sans perte de généralité, aucun axe ici considéré n'est du type `sibling`. En effet, même si un document t est donné sous une forme compressée, les parties des requêtes utilisant des axes horizontaux sont évaluées sur les chaînons qui représentent les suites des

nœuds frères, et ne subissent aucune compression par rapport à la représentation arborescente du document en question.

Commençons par quelques observations générales. Soient une requête Q, un trdag t, et son arbre équivalent $\widehat{t}$. La REPONSE $R^{\widehat{t}}_Q \subseteq Nodes_{\widehat{t}}$ à Q sur l'arbre $\widehat{t}$, est en général un sous–ensemble de la REPONSE $R^t_Q \subseteq Nodes_t$ à Q sur une compression t de $\widehat{t}$. Plus formellement, soient la surjection de compression (voir page 42 pour définition) $\mathbf{c}\colon Nodes_{\widehat{t}} \to Nodes_t$, un nœud $\alpha \in Nodes_{\widehat{t}}$, et $v \in Nodes_t$, tels que $\mathbf{c}(\alpha) = v$. Si $\alpha \in R^{\widehat{t}}_Q$, alors $v \in R^t_Q$. On peut alors écrire que $\mathbf{c}(R^{\widehat{t}}_Q) \subseteq R^t_Q$. La remarque suivante dit que l'inclusion réciproque n'est pas toujours vérifiée.

Remarque 4.2. *Soient t, $\widehat{t}$, $\mathbf{c}\colon Nodes_{\widehat{t}} \to Nodes_t$, et Q, comme plus haut. Considérons un nœud v de t, et $\alpha \in Nodes_{\widehat{t}}$, tels que $\mathbf{c}(\alpha) = v$.*

- *Le nœud v peut être sélectionné par Q sur t, sans que α soit dans la REPONSE à Q sur $\widehat{t}$ (voir l'Exemple 4.5, $\alpha = 1$ ou $\alpha = 3$).*
- *La requête Q peut avoir une REPONSE non–vide sur t, mais admettre une REPONSE vide sur l'arbre équivalent $\widehat{t}$ (voir l'Exemple 4.6).*

Exemple 4.6. *Soit la requête composée $Q =$* `//*[parent::`a *and* `parent::`b`]`. *La REPONSE à Q sur le trdag totalement compressé $t = f(a(c), b(c))$ contient un nœud (celui portant le nom c). Pourtant, si on évalue Q sur l'arbre $\widehat{t}$ équivalent à t, on obtient la REPONSE vide, car sur un arbre chaque nœud a au plus un père, et la requête Q en demande deux différents.*

Notons que les situations mentionnées dans la Remarque 4.2 peuvent arriver seulement si la requête Q comporte au moins un axe montant, c.–à–d., `parent` ou `ancestor`. En effet, les relations définies par ces axes sont moins triviales sur les trdags que sur les arbres. Cette observation nous donne le résultat suivant :

Lemme 4.1. *Soit* `axis` *un des axes* `self`, `child`, `descendant`. *Considérons une requête de base $Q =$* `//*[axis::`σ`]`, *l'automate A_Q correspondant, un trdag t donné, et son arbre équivalent $\widehat{t}$. Notons par $\mathbf{c}\colon Nodes_{\widehat{t}} \to Nodes_t$ la surjection de compression entre $\widehat{t}$ et t. Soient v un nœud de t, et α un nœud sur $\widehat{t}$, tels que $\mathbf{c}(\alpha) = v$. Le run de priorité maximale de l'automate A_Q sur $\mathcal{D}_t$, et le run de priorité maximale de l'automate A_Q sur $\mathcal{D}_{\widehat{t}}$, assignent aux nœuds v et α respectivement, la même ll–paire. En particulier, le nœud v est sélectionné si et seulement si, le nœud α est aussi sélectionné.*

Preuve. On sait que le run de priorité maximale est une fonction construite d'une façon top–down, qui est conforme avec la sémantique représentée dans la Table 4.1. Il suffit d'apercevoir que $\mathbf{c}(Nodes_{\widehat{t}|_\alpha}) = Nodes_{t|_v}$, c.–à–d., que l'image des descendants de α sur $\widehat{t}$, est composé des nœuds descendants de v sur t.

□

Soit un trdag t, son arbre équivalent $\widehat{t}$, et une requête Q contenant au moins un axe montant. La Remarque 4.2 et le Lemme 4.1 impliquent que, pour pouvoir déduire $R^{\widehat{t}}_Q$ à partir de R^t_Q, il faut modifier certaines transitions des automates correspondants aux requêtes *montantes* `//*[parent::`σ`]` et `//*[ancestor::`σ`]`, pour

les rendre compatibles avec la sémantique des axes `parent` et `ancestor` sur l'arbre. On appellera les automates ainsi obtenus *automates révisés*. On définira à nouveau la notion du run de priorité maximale pour ces automates révisés. Après cette révision, les automates ne sélectionneront pas de nœuds d'un rlag, mais seulement un sous–ensemble de positions, grâce auxquelles le nœud en question est sélectionné.

Soit t un trdag donné, le rlag $\mathcal{D}_t$ correspondant, et l'arbre $\widehat{t}$ équivalent à t. Avant de passer à la construction des automates révisés, essayons de comprendre le lien entre les positions de t, de $\mathcal{D}_t$ et celles de $\widehat{t}$. D'après la Section 4.2 on sait que les arêtes parallèles sur t, sont représentées par une seule arête sur $\mathcal{D}_t$. Une arête donnée sur $\mathcal{D}_t$ peut alors en général représenter plusieurs arêtes du t, qui elles–mêmes représentent parfois plusieurs arêtes de $\widehat{t}$ chacune (voir la Figure 4.1). Pour trouver

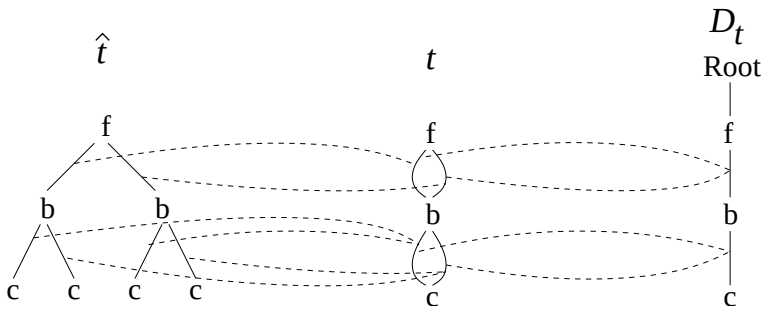
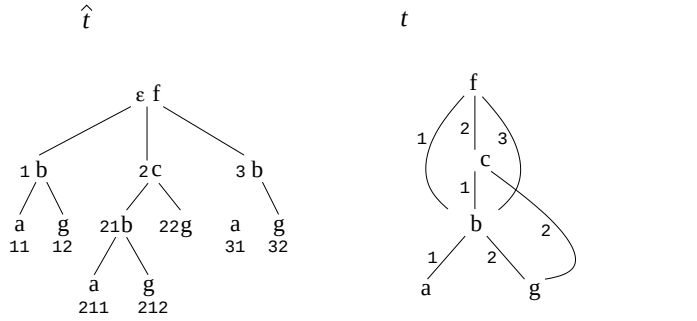


Figure 4.17. Correspondance entre les arêtes de l'arbre $\widehat{t}$, du trdag t et du rlag $\mathcal{D}_t$

toutes les positions représentées par un nœud de $\mathcal{D}_t$, assignons d'abord, à toute arête de t, un entier étant le numéro de cette arête en tant que fils du nœud à partir duquel elle sort. Ensuite, à chaque arête de $\mathcal{D}_t$ on assigne un symbole (on ne le fait pas pour l'arête reliant la racine A_0 avec le nœud A_1, car elle ne correspond à aucune arête sur t). Si X est le symbole attaché à l'arête (v, u) sur $\mathcal{D}_t$, alors X correspond à l'ensemble des entiers qui sont les numéros d'arêtes représentés par (v, u) sur t. Pour représenter symboliquement toutes les positions correspondantes à un nœud donné v de $\mathcal{D}_t$, il suffit de suivre tous les chemins allant de la racine vers v, et construire des mots symboliques correspondants. On appellera ces mots *positions symboliques*. Dans la suite de cette section, le mot position sera utilisé pour désigner une telle position symbolique, et $pos_{\mathcal{D}_t}(v)$ représentera l'ensemble des positions symboliques au nœud v. On illustre la construction des positions symboliques dans l'exemple suivant.

Exemple 4.7. *Considérons le trdag t, le rlag $\mathcal{D}_t$ associé, et l'arbre $\widehat{t}$ équivalent à t, qui sont représentés sur la Figure 4.18 (pour plus de clarté, on représente les nœuds de $\mathcal{D}_t$, par leurs llabs). Le symbole X représente les entiers* 1 *et* 3 *— première et troisième arête sortante du nœud f sur t. Les symboles Y, K et L correspondent à l'entier* 2*, et les symboles Z, T représentent l'entier* 1*. Une seule feuille de $\mathcal{D}_t$, ayant comme positions symboliques XK, YZK et YL, représente alors quatre positions du $\widehat{t}$:* $12, 32$ *codées par XK,* 212 *codée par YZK, et* 22 *codée par YL.*


$\hat{t}$
ε f
1 b
2 c
3 b
a
11
g
12
21 b
22 g
a
31
g
32
a
211
g
212
t
f
1
2
3
c
1
b
2
1
2
a
g
D_t

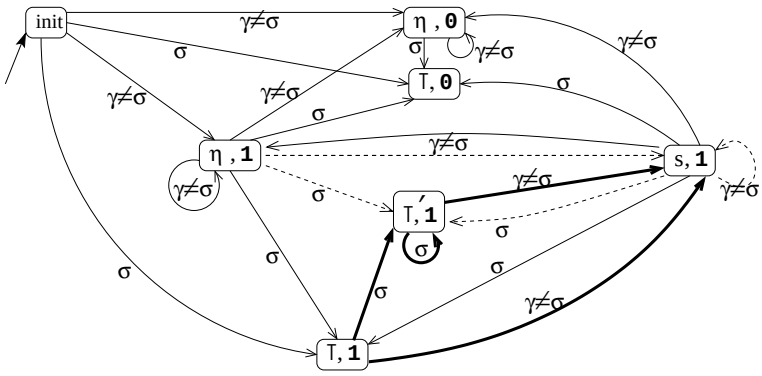


Figure 4.20. Automate révisé RA_Q pour $Q = //*[\texttt{parent::}\sigma]$

Exemple 4.8. *Considérons la transition* $((\eta, 1), \gamma) \rightarrow (s, 1)$ *sur l'automate révisé pour* $Q = //*[\texttt{parent::}\sigma]$. *Soient deux nœuds* v *et* u *d'un rlag* $\mathcal{D}_t$, *tels que* v *est le père de* u, *le llab à* u *est différent de* σ, *et le run de priorité maximale de* A_Q *(non–révisé) évaluant la requête* $Q = //*[\texttt{parent::}\sigma]$ *(voir la Figure 4.7) sur* $\mathcal{D}_t$, *a déjà assigné à* v *l'état* $(\eta, 1)$ *(ce qui implique entre autres que* $llab(v)$ *n'est pas* σ*). Le run de* A_Q *peut alors assigner au nœud* u *l'état sélectionnant* $(s, 1)$. *Bien sûr, une telle situation aura lieu si et seulement si* u *possède un autre nœud père (différent de* v*) dont le llab est* σ. *On voit bien que cela ne peut jamais arriver sur un arbre, car un nœud donné sur un arbre possède toujours au plus un père. C'est pour cela que dans l'automate révisé* RA_Q, *cette transition est du type "forget".*

Le deuxième ensemble de transitions distinguées, est composé de toutes les transitions partant de $(\top, 1)$, et $(\top', 1)$. Ce sont des transitions applicables à partir d'un nœud qui porte le llab σ. On les appelle transitions *restore*, et on les marque en gras.

On donne maintenant la définition formelle du run de priorité maximale d'un automate révisé RA_Q sur un rlag $\mathcal{D}_t$.

Définition 4.6. *Soient* $Q = //*[\texttt{axis::}\sigma]$ *une requête de base montante, les automates* A_Q *(Section 4.3.1) et* RA_Q *correspondants, et un rlag* $\mathcal{D}_t$ *donné. Le run de priorité maximale de* RA_Q *sur* $\mathcal{D}_t$ *est une fonction*

$$r \colon Nodes_{\mathcal{D}_t} \rightarrow States_{RA_Q} \times 2^{Pos_{\mathcal{D}_t}} \times 2^{Pos_{\mathcal{D}_t}},$$

qui satisfait les conditions suivantes :

- $\pi_1(r) = r_{A_Q}$, *où* r_{A_Q} *est le run de priorité maximale de l'automate* A_Q *sur* $\mathcal{D}_t$*; c.–à–d., pour tout nœud* v *de* $\mathcal{D}_t$, *l'état assigné par* r *à* v *est exactement le même que l'état qui serait assigné par le run de priorité maximale de l'automate* A_Q *sur* $\mathcal{D}_t$,
- *pour tout nœud* v, *si* $r(v) = ((\ell, x), P(v), \overline{P(v)})$, *alors* $P(v) \dot{\cup} \overline{P(v)} = pos_{\mathcal{D}_t}(v)$, *c.–à–d., les ensembles* $P(v)$ *et* $\overline{P(v)}$ *forment toujours une partition de l'ensemble des positions représentées par le nœud* v,
- *si* $label(v) = (Root, -)$, *alors* $\pi_3(r(v)) = \overline{P(v)} = \emptyset$.

De plus, soient deux nœuds v et u de $\mathcal{D}_t$, tels que u est un enfant de v, et une position $\alpha \in pos_{\mathcal{D}_t}(v)$. Supposons que $r(v) = ((\ell, x), P(v), \overline{P(v)})$.

- *si $\alpha \in P(v)$, et pour passer de v à u le run utilise une transition habituelle ou "restore", alors pour toutes les positions $\alpha.X \in pos_{\mathcal{D}_t}(u)$, on a $\alpha.X \in P(u)$,*
- *si $\alpha \in \overline{P(v)}$, et pour passer de v à u le run utilise une transition habituelle ou "forget", alors pour toutes les positions $\alpha.X \in pos_{\mathcal{D}_t}(u)$, on a $\alpha.X \in \overline{P(u)}$,*
- *si $\alpha \in \overline{P(v)}$, et pour passer de v à u le run utilise une transition "restore", alors pour toutes les positions $\alpha.X \in pos_{\mathcal{D}_t}(u)$, on a $\alpha.X \in P(u)$.*

La REPONSE à Q sur l'arbre $\widehat{t}$ est alors composée de toutes les positions représentées par les ensembles $P(v)$, pour les nœuds v de $\mathcal{D}_t$, tels que $\pi_1(r(v)) = (l', 1)$. En d'autres termes, soit un nœud v de $\mathcal{D}_t$, auquel le run de RA_Q a assigné l'état sélectionnant $(l', 1)$. L'ensemble de nœuds correspondants à v sur $\widehat{t}$, qui constituent la REPONSE à Q sur $\widehat{t}$, est composé seulement par ces nœuds qui sont représentés par les positions symboliques qui se trouvent dans $\pi_2(r(v)) = P(v)$. On illustre la construction du run d'un automate révisé, dans l'exemple ci–dessous.

Exemple 4.9. *On veut évaluer la requête $Q =$ `//*[parent::a]`, sur le trdag t représenté sur la Figure 4.21 à gauche. Puisque sur t il n'y a pas d'arêtes parallèles, le trdag t et son rlag associé $\mathcal{D}_t$ sont isomorphes. Par suite, on présente les runs de A_Q et RA_Q directement sur t.*

Les arbres au milieu et à droite de la Figure 4.21 montrent que l'évaluation de Q sur le trdag compressé t, en utilisant l'automate non révisé A_Q, ne donne pas la même REPONSE que l'évaluation de Q sur l'arbre équivalent à t.

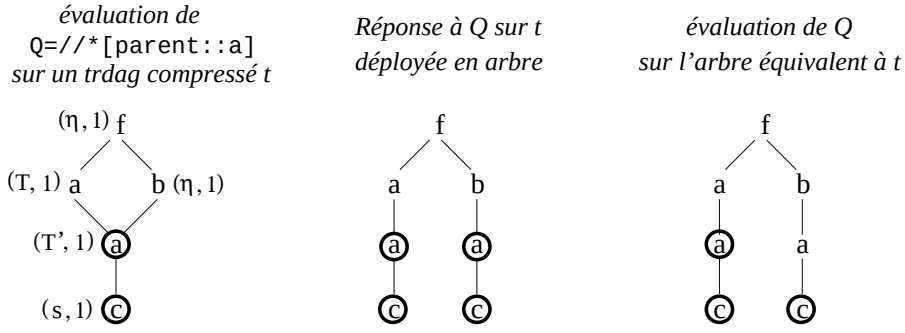


Figure 4.21. Résultat d'évaluation de $Q =$ `//*[parent::a]` à l'aide de l'automate non révisé

Sur la Figure 4.22, on présente l'évaluation de Q sur t, en utilisant l'automate révisé RA_Q. Notons par v le nœud qui représente les positions 11 *et* 21*, et par u l'unique feuille de t.*

- *Pour passer du nœud représentant la position* 2 *au nœud v, le run de l'automate RA_Q a utilisé la transition $((\eta, 1), a) \to (\top', 1)$, qui est du type "forget". Conformément à la définition du run, la position* 21 *se trouve alors dans l'ensemble de positions "oubliées" $\overline{P(v)}$ (ce qu'on représente sur la Figure 4.22 par une barre au dessus de* 21*). Bien que l'état assigné à v est sélectionnant, la position* 21 *ne constitue pas une réponse à Q.*

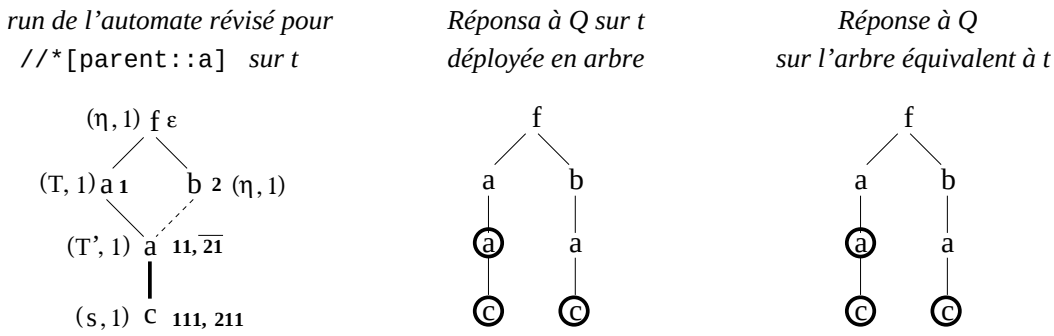


Figure 4.22. Résultat d'évaluation de $Q = //*[\texttt{parent::}a]$ à l'aide de l'automate révisé

- *Pour passer du nœud v à la feuille u, le run de l'automate RA_Q a utilisé la transition $((\top', 1), \gamma \neq a) \rightarrow (s, 1)$, qui est du type "restore". Par conséquent, toutes les positions représentées par u se trouvent dans l'ensemble $P(u)$, et constituent des réponses à Q sur t (même la position* 211 *dont le préfixe (*21*) à été "oublié" au nœud v).*

Finalement, la REPONSE à Q sur t est composée des positions $11, 111, 211$*, ce qui correspond à la REPONSE à Q sur l'arbre équivalent à t.*

4.7.1. Requêtes composées via les automates révisés

Il reste à montrer comment déduire (à partir de l'évaluation sur D_t) la REPONSE à une requête composée Q sur l'arbre équivalent $\widehat{t}$. Notons que, si Q contient au moins un axe montant, alors on doit utiliser les automates révisés.

Dans la Remarque suivante, on définit les automates révisés pour les axes différents de `parent` et `ancestor`.

Remarque 4.3. *Soit une requête de base $Q = //*[\texttt{axis::}\sigma]$, où* `axis` *n'est ni* `parent` *ni* `ancestor`*. Dans cette situation, l'automate révisé RA_Q n'est rien d'autre que l'automate A_Q (défini dans la Section 4.3), où toutes les transitions sont supposées du type habituel (pas de transitions "forget" ni "restore"). De plus, notons par r_{A_Q} le run de priorité maximale de A_Q sur un rlag donné $\mathcal{D}_t$, si* `axis` *est un axe vertical, et la fonction $\widehat{r}$ définie à la page 55, si* `axis` *est un axe horizontal. Le run de priorité maximale r de l'automate révisé RA_Q sur $\mathcal{D}_t$ est défini comme suit : pour tout $v \in Nodes_{\mathcal{D}_t}$, $r(v) = (r_{A_Q}(v), pos_{\mathcal{D}_t}(v), \emptyset)$.*

Commençons par une requête du type $Q = //*[\texttt{axis}_1\texttt{::}\sigma_1 \ conn \ \texttt{axis}_2\texttt{::}\sigma_2]$, où $conn \in \{and, or\}$. Considérons un trdag t, son arbre équivalent $\widehat{t}$, et le rlag $\mathcal{D}_t$ correspondant à t. Notons par Q_1 la requête $//*[\texttt{axis}_1\texttt{::}\sigma_1]$ et par Q_2 la requête $//*[\texttt{axis}_2\texttt{::}\sigma_2]$. Soit r_i où $i \in \{1, 2\}$, le run de priorité maximale de l'automate révisé RA_{Q_i} sur le rlag $\mathcal{D}_t$. Voici comment on modifie les fonctions AND et OR de la section 4.6, pour obtenir les fonctions $RAND$ et ROR, qu'on appellera *révisées*,

et qui tiendront compte des ensembles de positions réellement sélectionnées par la requête en question :

$$RCONN\colon Nodes_{\mathcal{D}_t} \to \{init, (s,1), (\eta,1), (\eta,0)\} \times 2^{Pos_{\mathcal{D}_t}} \times 2^{Pos_{\mathcal{D}_t}},$$

si $conn = and$, alors $RCONN = RAND$, et pour tout $v \in Nodes_{\mathcal{D}_t}$ on a :

$$RAND(v) = \begin{cases} (init, \emptyset, \emptyset), & \text{si } \pi_1(r_1(v)) = \pi_1(r_2(v)) = init, \\ ((s,1), P_\wedge(v), \overline{P_\wedge(v)}), & \text{si } \pi_1(r_1(v)) = \pi_1(r_2(v)) = (l',1), \\ ((\eta,0), pos_{\mathcal{D}_t}(v), \emptyset), & \text{si } \pi_1(r_1(v)) = (l,0) \text{ ou } \pi_1(r_2(v)) = (l,0), \\ ((\eta,1), pos_{\mathcal{D}_t}(v), \emptyset), & \text{ailleurs,} \end{cases}$$

$$\text{avec } P_\wedge(v) = \pi_2(r_1(v)) \cap \pi_2(r_2(v)), \text{ et } \overline{P_\wedge(v)} = \pi_3(r_1(v)) \cup \pi_3(r_2(v));$$

si $conn = or$, alors $RCONN = ROR$, et pour tout $v \in Nodes_{\mathcal{D}_t}$ on a :

$$ROR(v) = \begin{cases} (init, \emptyset, \emptyset), & \text{si } \pi_1(r_1(v)) = \pi_1(r_2(v)) = init, \\ ((s,1), P_\vee(v), \overline{P_\vee(v)}), & \text{si } \pi_1(r_1(v)) = (l',1) \text{ ou } \pi_1(r_2(v)) = (l',1), \\ ((\eta,0), pos_{\mathcal{D}_t}(v), \emptyset), & \text{si } \pi_1(r_1(v)) = \pi_1(r_2(v)) = (l,0), \\ ((\eta,1), pos_{\mathcal{D}_t}(v), \emptyset), & \text{ailleurs,} \end{cases}$$

$$\text{avec } P_\vee(v) = \pi_2(r_1(v)) \cup \pi_2(r_2(v)), \text{ et } \overline{P_\vee(v)} = \pi_3(r_1(v)) \cap \pi_3(r_2(v)).$$

Il n'est pas difficile de vérifier que les fonctions $RAND$ et ROR sont bien définies — en particulier, pour chaque nœud v de $\mathcal{D}_t$, les ensembles $P_\wedge(v)$ et $\overline{P_\wedge(v)}$, ainsi que $P_\vee(v)$ et $\overline{P_\vee(v)}$, forment bien les partitions de $pos_{\mathcal{D}_t}(v)$ (puisque la racine fictive *Root* de $\mathcal{D}_t$, ne présente aucune position de t, on suppose que $pos_{\mathcal{D}_t}(Root) = \emptyset$). Soient un nœud v de $\mathcal{D}_t$, et une position α de $\widehat{t}$, telle que $\alpha \in pos_{\mathcal{D}_t}(v)$. Le nœud de $\widehat{t}$ à la position α appartient à la REPONSE à Q sur $\widehat{t}$, si et seulement si :

$$\pi_1(RCONN(v)) = (s,1) \quad \text{et} \quad \alpha \in P_{conn}(v).$$

Il reste à montrer comment on procède dans le cas d'une requête imbriquée, qui utilise au moins un axe `parent` ou `ancestor`. Soit

$$Q = \texttt{//*[axis}_1\texttt{::*[axis}_2\texttt{::*[...[axis}_{k-1}\texttt{::*[}S_k\texttt{]]...]]]}$$

une telle requête, où $k > 1$, et S_k est soit de la forme `axis::`σ, soit de la forme (`axis::`σ *conn* `axis'::`σ'). Pour tout i, $1 \leq i \leq k-1$, notons par S_i l'expression `axis`$_i$`::*`, et posons $\sigma_i = *$. Comme on a vu dans la Section 4.6, pour évaluer la requête Q sur un document donné t, on commence par trouver la REPONSE correspondante à la partie S_k sur $\mathcal{D}_t$, ensuite on relabelle le rlag $\mathcal{D}_t$ conformément à cette REPONSE, et on passe à l'évaluation de la partie correspondante à S_{k-1} sur le rlag ainsi relabelé. Après avoir traversé d'une telle façon (de droite à gauche) la requête toute entière, on obtient la REPONSE à Q sur t. Dans le cas des automates révisés on procédera de la même façon récursive décroissante sur $i \in \{k, \ldots, 1\}$, la seule chose à modifier est la définition de la fonction de relabeling $label_i$, qui sert à relabeler le rlag $\mathcal{D}_t$ après avoir évalué la partie S_i. Considérons l'indice $i \in \{1, \ldots, k\}$,

et supposons qu'on a déjà évalué la partie correspondante à S_i sur le rlag $r_{i+1}(\mathcal{D}_t)$ (on reprend la notation de la Section 4.6, en supposant que $r_{k+1}(\mathcal{D}_t) = \mathcal{D}_t$). La fonction de relabeling $rlabel_i$, qui doit être utilisée dans le cas des automates révisés, est alors définie comme suit :

$$rlabel_i \colon Nodes_{\mathcal{D}_t} \to \mathcal{P}(\{(Root, -)(s,1), (\eta,1), (\eta,0)\} \times 2^{Pos_{\mathcal{D}_t}})$$

$$rlabel_i(v) = \begin{cases} ((Root,-), pos_{\mathcal{D}_t}(v)), & \text{si } \pi_1(r_i(v)) = init, \\ ((s,1), \pi_2(r_i(v))), ((\eta,1), \pi_3(r_i(v))), & \text{si } \pi_1(r_i(v)) \in \{(s,1), (\top',1)\}, \\ ((\eta,1), pos_{\mathcal{D}_t}(v)), & \text{si } \pi_1(r_i(v)) \in \{(\eta,1), (\top,1)\}, \\ ((\eta,0), pos_{\mathcal{D}_t}(v)), & \text{si } \pi_1(r_i(v)) \in \{(\eta,0), (\top,0)\}, \end{cases}$$

avec :

- si $S_i = \texttt{axis}_i\texttt{::}\sigma_i$, et $\texttt{axis}_i$ est un axe vertical, alors la fonction r_i utilisé dans la définition de $rlabel_i$ est le run de priorité maximale de l'automate RA_Q correspondant à la requête $Q_i = \texttt{//*[axis}_i\texttt{::}\sigma_i\texttt{]}$, sur le rlag $r_{i+1}(\mathcal{D}_t)$.
- si $S_i = \texttt{axis}_i\texttt{::}\sigma_i$, et $\texttt{axis}_i$ est un axe horizontal, alors la fonction r_i représente la fonction $rev(\widehat{r})$, qui est la fonction $\widehat{r}$ (voir page 55) révisée, définie de la façon suivante : pour tout $v \in Nodes_{\mathcal{D}_t}$, $rev(\widehat{r})(v) = (\widehat{r}(v), pos_{\mathcal{D}_t}(v), \emptyset)$.
- si $i = k$ et $S_k = (\texttt{axis}_k\texttt{::}\sigma_k \ conn \ \texttt{axis'}_k\texttt{::}\sigma'_k)$, alors la fonction r_k représente la fonction révisée $RAND$ ou ROR conformément à $conn$.

Après avoir évalué la partie S_i sur le rlag $r_{i+1}(\mathcal{D}_t)$, on relabelle ce dernier en utilisant la fonction révisée $rlabel_i$, et en produisant ainsi le rlag appelé $r_i(\mathcal{D}_t)$. Ensuite, on évalue la partie correspondante à S_{i-1}. Pour le faire, on fait courir sur le rlag $r_i(\mathcal{D}_t)$ l'automate révisé correspondant à la requête $\texttt{//*[axis}_{i-1}\texttt{::}s\texttt{]}$. Son rôle est de sélectionner ces nœuds v de $r_i(\mathcal{D}_t)$, pour lesquelles la première composante de $rlabel_i(v)$ est $(s,1)$. Remarquons que, sur le rlag $r_i(\mathcal{D}_t)$, on peut avoir des nœuds qui ont deux labels. C'est le cas des nœuds v qui ont été sélectionnés par la partie S_i, où $rlabel_i(v)$ est composé de $((s,1), \pi_2(r_i(v)))$ et $((\eta,1), \pi_3(r_i(v)))$. La sémantique d'un tel double label est la suivante :

- les positions représentées par $\pi_2(r_i(v)))$ correspondent à des réponses à S_i sur $\widehat{t}$, et ont le label $(s,1)$;
- celles représentées par $\pi_3(r_i(v))$ ne constituent pas des réponses à S_i sur $\widehat{t}$, et par suite obtiennent le label $(\eta,1)$.

Pendant l'évaluation de la partie S_{i-1} sur la rlag $r_i(\mathcal{D}_t)$ il faut alors voir des nœuds v ayant un double label, comme deux nœuds séparés : le premier ayant le label $(s,1)$, représentant des positions sélectionnées; et le deuxième ayant le label $(\eta,1)$, représentant des positions non–sélectionnées. Par suite, l'automate évaluant la partie S_{i-1} appliquera deux transitions différentes à partir de tels nœuds v (voir l'Exemple 4.10).

Après une telle évaluation récursive de toutes les parties S_i, $i \in \{k, \ldots, 1\}$, on obtient le rlag $r_1(\mathcal{D}_t)$ qui représente la REPONSE à notre requête imbriquée Q. Cette REPONSE est composée par des positions auxquelles la fonction $rlabel_1$ à assigné le label $(s,1)$. Bien évidemment, elle correspond à la REPONSE à la requête Q sur l'arbre $\widehat{t}$ équivalent à t. On illustre l'évaluation d'une requête composée à l'aide des automates révisés, sur l'exemple ci–dessous.

Exemple 4.10. *Dans cet exemple, on évalue la requête*

$$//*[\texttt{ancestor::}b[\texttt{parent::}c]],$$

sur le document compressé t, représenté au milieu de la Figure 4.18. La forme standardisée de notre requête est

$$Q = //*[\underbrace{\texttt{ancestor::}*}_{S_1}][\underbrace{\texttt{self::}b \ \textit{and} \ \texttt{parent::}c}_{S_2}];$$

et la REPONSE à Q est composée de tous les nœuds qui ont un ancêtre b avec un père c. Pour plus de clarté, on représente les nœuds du rlag $\mathcal{D}_t$ à l'aide de leurs llabs. On commence par trouver la REPONSE à la requête $//*[S_2]$= //[self::b and parent::c], sur $\mathcal{D}_t$. Tout d'abord on évalue les requêtes //*[self::b] et //*[parent::c] en parallèle, en utilisant des automates révisés, et ensuite on applique la fonction révisée RAND (la Figure 4.23). On obtient ainsi le rlag $RAND(\mathcal{D}_t)$*

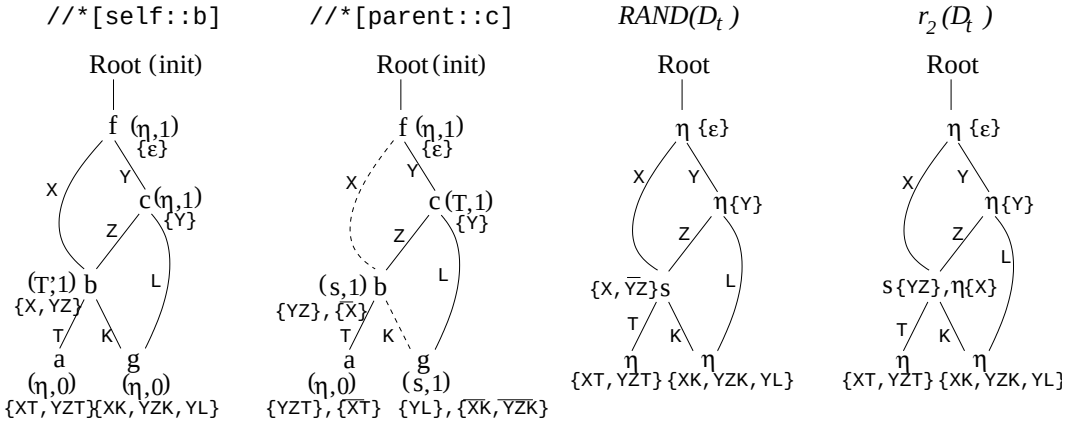


Figure 4.23. Évaluation de //*[self::b *and* parent::c] à l'aide des automates révisés

(le troisième sur la figure mentionnée). On le relabelle conformément à la fonction de relabeling $rlabel_2$, et on obtient la rlag $r_2(\mathcal{D}_t)$ (quatrième sur la Figure 4.23). Remarquons que le nœud du $r_2(\mathcal{D}_t)$, qui représente des positions X et YZ, reçoit maintenant deux labels : $(s,1)$ correspondant à la position YZ, et $(\eta,1)$ correspondant à la position X.

Maintenant, on évalue la requête //[ancestor::s] *sur le rlag $r_2(\mathcal{D}_t)$ (Figure 4.24 à gauche). Notons que deux transitions différentes sont utilisées pour passer du nœud représentant les positions X et YZ à la feuille correspondant à XT et YZT. En effet,*

- *pour passer de X vers XT, le run utilise la transition $((\eta,1),\eta) \to (s,1)$ du type "forget",*
- *et pour passer de YZ vers YZT, le run utilise la transition $((\top,1),\eta) \to (s,1)$ qui permet de sélectionner la position YZT.*

Idem pour passer vers l'autre feuille. Finalement, on relabelle le rlag $r_2(\mathcal{D}_t)$, conformément à la fonction de relabeling $rlabel_1$, pour obtenir le rlag $r_1(\mathcal{D}_t)$ (Figure 4.24 à droite). Ses deux feuilles représentent cinq positions symboliques, mais seulement deux d'entre elles, YZT et YZK, ont le llab s, et représentent la REPONSE à Q sur

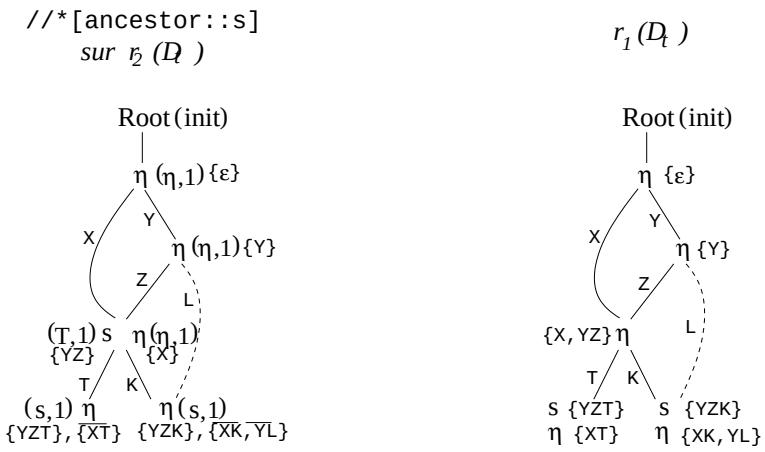


Figure 4.24. Évaluation de //∗[self::b] et //∗[parent::c] à l'aide des automates révisé

t. La position symbolique YZT de $\mathcal{D}_t$ représente la position 211, *et YZK représente la position* 212. *Comme on peut le voir sur la Figure 4.25, ces positions forment la REPONSE à Q sur l'arbre $\widehat{t}$ équivalent à t.*

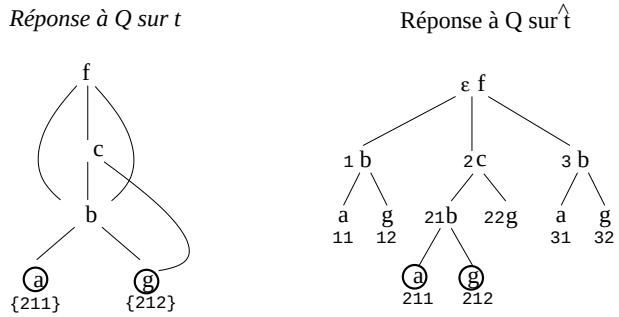


Figure 4.25. REPONSE à Q sur t et sur son arbre équivalent $\widehat{t}$

Chapitre 5

Inclusion de patterns via la réécriture

Nous présentons ici une approche, basée sur des techniques de réécriture, qui permet de résoudre le problème d'inclusion des schémas de requêtes (pattern containment) pour le fragment XP(/,//,[ ],$*$) de XPath ([67, 77]). Nous définissons un système de réécriture basé sur la sémantique de l'inclusion de patterns du segment XP(/,//,[ ],$*$), et montrons que pour deux patterns donnés P et Q, P est inclus dans Q si et seulement si, il est possible de réécrire P vers Q en utilisant les règles de ce système. L'approche de ce chapitre sera publiée dans [50].

5.1. Inclusion de patterns

Fixons un alphabet Σ. On considère ici le fragment XP(/,//,[ ],$*$) composé des expressions de XPath ne contenant que les axes `child` et `descendant`, et où on autorise l'utilisation des filtres qualificatifs, du symbole '$*$' (don't–care) de XPath, et des symboles de Σ. Tout élément de XP(/,//,[ ],$*$) est une requête qui peut être représentée par un graphe arborescent appelé *pattern unaire* ([6]), ayant :

- deux types d'arêtes : enfant et descendant,
- les nœuds étiquetés par des symboles de $\Sigma \cup \{*\}$,
- un nœud labelé 's' correspondant au nœud sélectionné par la requête.

Pour un pattern donné P, on dénotera par $Edges_{\downarrow}(P)$ et $Edges_{\Downarrow}(P)$ respectivement l'ensemble des arêtes du type enfant, et celui des arêtes du type descendant. La notation $Nodes_P$ sera utilisée pour désigner l'ensemble des nœuds du pattern P. Par $name_P(v)$ on dénotera l'étiquette au nœud v de P. Le pattern de la Figure 5.1, où les arêtes simples sont ceux du type child et les arêtes doubles sont du type descendant, représente la requête $P = a//b[./b/c/d]/c[./*//d]$. On peut étendre la notion de

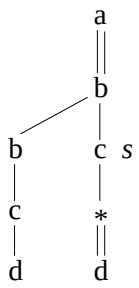


Figure 5.1. Pattern P représentant la requête $P = a//b[./b/c/d]/c[./*//d]$

pattern unaire, à celle de *pattern n–aire*, où on a n nœuds labelés $s_1, \ldots, s_n$. De tels patterns peuvent servir pour modéliser les requêtes n–aires (sélectionnant un ensemble de n–uplètes de nœuds). Miklau et Suciu introduisent dans [6] la notion d'un *pattern booléen* qui ne contient pas de nœuds labelés. Ils montrent qu'à l'aide de tels patterns booléens, on peut représenter exactement la même information qu'en utilisant les patterns n–aires. Par suite (sans mention contraire), tous les patterns considérés ici seront booléens, et on les appellera tout simplement patterns.

Soient un pattern P et un arbre XML t. On appelle *plongement* de P dans t toute fonction $\mathbf{e}$: $Nodes_P \rightarrow Nodes_t$, qui satisfait les conditions suivantes :

- $\mathbf{e}$ préserve la racine : $\mathbf{e}(root_P) = root_t$;
- $\mathbf{e}$ préserve les noms : $\forall u \in Nodes_P$, $name_P(u) = *$, ou $name_P(u) = name_t(\mathbf{e}(u))$;
- $\mathbf{e}$ préserve la relation *child* : $\forall (u,v) \in Edges_{\downarrow}(P)$, $(\mathbf{e}(u), \mathbf{e}(v)) \in Edges_t$;
- $\mathbf{e}$ préserve la relation *descendant* : $\forall (u,v) \in Edges_{\Downarrow}(P)$, $(\mathbf{e}(u), \mathbf{e}(v)) \in (Edges_t)^+$,

où $(Edges_t)^+$ est la clôture transitive de $Edges_t$ vu comme une relation.

Définition 5.1. *Un document (arbre) XML t est appelé modèle d'un pattern P ssi il existe un plongement de P dans t.*

la Figure 5.2 illustre la notion de modèle d'un pattern.

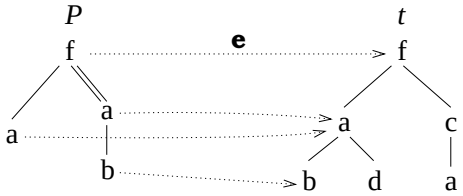


Figure 5.2. Pattern P, un modèle t de P, et un plongement de P dans t

Définition 5.2. *Soient deux patterns P et Q. On dit que P est inclus dans Q ($P \subseteq Q$) ssi tout modèle de P est également un modèle de Q. Les patterns P et Q sont dits équivalents ($P \equiv Q$) ssi $P \subseteq Q$ et $Q \subseteq P$.*

Sur la Figure 5.3 on présente deux patterns équivalents P et Q.

Figure 5.3. Deux patterns équivalents P et Q

Les auteurs de [6] prouvent que le problème d'inclusion de patterns pour le fragment XP(/,//,[ ],$*$) est CoNP–complet. Ils donnent également une condition suffisante pour résoudre le problème d'inclusion de patterns. Pour cela, ils introduisent la

notion d'homomorphisme d'un pattern Q vers un pattern P. Soient deux patterns P et Q. On appelle *homomorphisme* de Q vers P une fonction φ: $Nodes_Q \rightarrow Nodes_P$, qui satisfait les conditions suivantes :

- φ préserve la racine : $\varphi(root_Q) = root_P$;
- φ préserve les noms : $\forall u \in Nodes_Q$, $name_Q(u) = *$, ou $name_Q(u) = name_P(\varphi(u))$;
- φ préserve la relation *child* : $\forall (u,v) \in Edges_{\downarrow}(Q)$, $(\varphi(u), \varphi(v)) \in Edges_{\downarrow}(P)$;
- φ préserve la relation *descendant* : $\forall (u,v) \in Edges_{\Downarrow}(Q)$, $(\varphi(u), \varphi(v)) \in (Edges_{\downarrow}(P) \cup Edges_{\Downarrow}(P))^+$.

Les auteurs de [6] prouvent que, s'il existe un homomorphisme de Q vers P, alors $P \subseteq Q$. Ils présentent un algorithme qui vérifie en temps $\mathcal{O}(|P||Q|)$, s'il existe un homomorphisme de Q vers P (où $|P|$ est le nombre d'arêtes du pattern P). La Figure 5.4 présente deux exemples de patterns P et Q, tels que $P \subseteq Q$, ainsi que des homomorphismes φ correspondants de Q vers P. Néanmoins, l'existence d'un

Figure 5.4. Patterns P et Q, tels que $P \subseteq Q$ et un homomorphisme φ de Q vers P

homomorphisme de Q vers P n'est pas une condition nécessaire pour que P soit inclus dans Q. Il suffit de considérer les patterns P et Q de la Figure 5.3 qui sont équivalents, mais pour lesquels il n'y a ni d'homomorphisme de P vers Q, ni de Q vers P. L'exemple suivant montre comment prouver l'inclusion $P \subseteq Q$, dans le cas où il n'y a pas d'homomorphisme de Q vers P.

Exemple 5.1. *La Figure 5.5 présente deux patterns P et Q satisfaisant $P \subseteq Q$, tels qu'il n'existe pas d'homomorphisme de Q vers P. Pour montrer que $P \subseteq Q$, il faut*

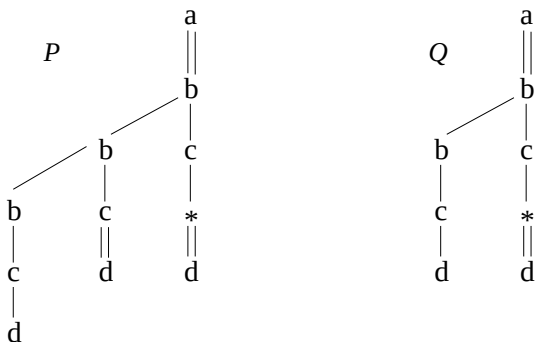


Figure 5.5. Patterns P et Q, tels que $P \subseteq Q$, mais pas d'homomorphisme de Q vers P

raisonner par cas. Soit t un modèle de P. L'arête $c//d$ sur la branche au milieu du pattern P, peut être réalisée sur t :

- *soit par une arête child c/d (comme sur la Figure 5.6),*

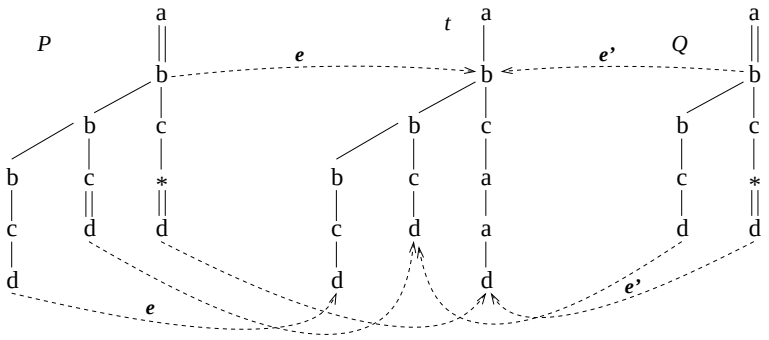


Figure 5.6. Modèle de P (et de Q), où $c//d$ est réalisé par c/d

- *soit par un chemin $c/*/\ldots/d$, de longueur ≥ 2 (voir la Figure 5.7).*

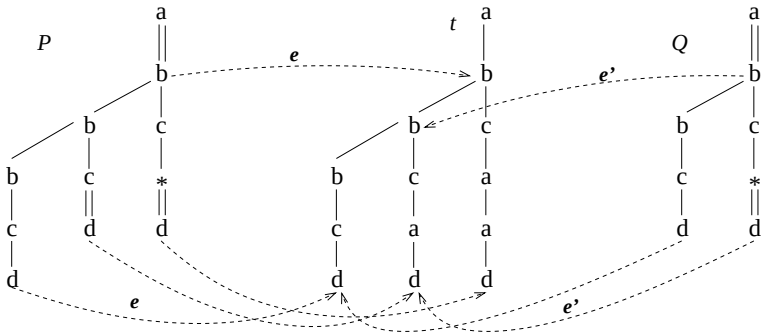


Figure 5.7. Modèle de P (et de Q), où $c//d$ est réalisé par le chemin $c/a/d$

L'analyse des deux cas mentionnés ci–dessus permet de conclure que tout modèle de P est aussi modèle de Q. Néanmoins, il est impossible de définir un homomorphisme général de Q vers P, car dans les deux cas considérés bien distincts, la branche $a//b/c/*//d$ (la plus à droite) de Q, correspond à différentes branches de P.

5.2. Réécriture et inclusion de patterns

Nous proposons ici une méthode, utilisant des techniques de réécriture, qui permet d'exprimer la notion d'homomorphisme d'une façon plus générale, pouvant inclure l'étude par cas. Nous construisons un ensemble $\mathcal{R}$ de règles de réécriture basées sur la sémantique d'inclusion de patterns du fragment XP(/,//,[ ],$*$), et montrons qu'étant donnés deux patterns P et Q, on a $P \subseteq Q$ si et seulement si on peut réécrire P vers Q en utilisant les règles de $\mathcal{R}$.

Pour bien définir les règles mentionnées, on introduit tout d'abord une définition formelle de pattern (alternative à celle utilisée dans la Section 5.1, où pattern est un arbre étiqueté avec deux types d'arêtes).

Définition 5.3. *Soit un alphabet donné Σ. On appelle pattern sur Σ toute expression P générée à partir de la grammaire présentée dans la Table 5.1, où $\omega \in \Sigma \cup \{*\}$.*

M	:	$\varepsilon \mid \downarrow\omega \mid \Downarrow\omega \mid MM$	// branche
S	:	$\emptyset \mid \{MS\} \mid S \cup S$	// ensemble de termes frères
P	:	ωMS	// patterns

Table 5.1. Grammaire pour générer des patterns

Les patterns au sens de la Définition 5.3 sont exactement ces expressions qui peuvent être représentées à l'aide d'un graphe raciné arborescent (comme dans la Section 5.1) et pouvant avoir deux types d'arêtes : simples pour $\downarrow$ (`child`), et doubles pour $\Downarrow$ (`descendant`). Par exemple, le graphe P (pattern au sens de [6]) à gauche de la Figure 5.5 correspond au pattern suivant (au sens de la Définition 5.3) :

$$P = a \Downarrow b\{\downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \Downarrow d\}, \downarrow c \downarrow * \Downarrow d\}.$$

On appelle *terme* toute expression du type M, S ou P générée par la grammaire de la Table 5.1, ainsi que toute disjonction finie $P_1 \vee P_2 \vee \cdots \vee P_n$ de patterns. Les termes du type M correspondent aux chemins linéaires sans branchement commençant par un symbole modal $\xi \in \{\downarrow, \Downarrow\}$; ceux du type S représentent un ensemble de termes ayant un père commun; et des termes du type P sont des patterns. Les termes du type M et S sont *non–racinés*, c.–à–d., ils ne commencent pas par un symbole de $\Sigma \cup \{*\}$; ceux du type P sont *racinés* – leur premier caractère est un symbole de $\Sigma \cup \{*\}$. Des termes en forme ε, P, ou $P_1 \vee \cdots \vee P_n$, où P, $P_1, \ldots, P_n$ sont des patterns, seront appelés *d–patterns* (patterns disjonctifs). On utilisera des d–patterns dans le raisonnement par cas, pour représenter, à l'aide d'un seul terme, les différents groupes de modèles d'un pattern donné.

Exemple 5.2. *Considérons les patterns*

$$P = f \downarrow * \Downarrow a \qquad et \qquad Q = f \Downarrow * \downarrow a$$

représentés sur la Figure 5.3. On sait que $P \equiv Q$, d'où en particulier $P \subseteq Q$, mais qu'il n'existe pas d'homomorphisme qui prouve cette inclusion. Grâce au système de réécriture $\mathcal{R}$ défini plus bas, on pourra réécrire P vers Q, et par conséquent prouver $P \subseteq Q$. L'idée est la suivante : Tout descendant est soit un enfant soit un descendant de profondeur ≥ 2, d'où, l'arête $* \Downarrow a$ de P peut être réalisée soit par une arête child $* \downarrow a$, soit par un chemin ayant au moins un nœud supplémentaire entre '$*$' et 'a', c.–à–d., $* \Downarrow * \downarrow a$. On réécrira alors le pattern P vers le d–pattern qui représente les cas considérés :

$$P = f \downarrow * \Downarrow a \longrightarrow_{\mathcal{R}} \; f \downarrow * \downarrow a \;\; \vee \;\; f \downarrow * \Downarrow * \downarrow a.$$

Ensuite, les composants du d–pattern obtenu seront réécrit parallèlement. Tout enfant, ainsi que tout descendant de profondeur ≥ 2 sont des cas particuliers d'un

descendant. Par suite, l'arête $f \downarrow *$ *peut être réécrite vers* $f \Downarrow *$, *idem pour le chemin* $f \downarrow * \Downarrow *$; *ce qui nous donne le d–pattern suivant :*

$$f \Downarrow * \downarrow a \quad \vee \quad f \Downarrow * \downarrow a.$$

Il sera finalement réécrit en Q, *car ses deux composantes représentent exactement le pattern* Q.

On présente maintenant notre ensemble $\mathcal{R}$ de règles de réécriture qui servent à réécrire les termes. Dans ces règles, les termes M, S, P (éventuellement ornés des primes ou des indices) sont comme dans la grammaire donnée dans la Table 5.1, D représente un d–pattern, $\sigma \in \Sigma$, et $\omega, \omega' \in \Sigma \cup \{*\}$:

1. $S \longrightarrow \emptyset$, $M \longrightarrow \varepsilon$ //ignorer un sous–terme;
2. $MS \longrightarrow \{MS\}$, $\{MS\} \longrightarrow MS$ // identification syntactique;
3. $M\sigma S \longrightarrow M * S$ //remplacer un symbole de Σ par '$*$' de XPath;
4. $\downarrow \omega S \longrightarrow \Downarrow \omega S$ //tout enfant est également un descendant;
5. $\xi\omega\xi'\omega' S \longrightarrow \Downarrow \omega' S$, où $\xi, \xi' \in \{\downarrow, \Downarrow\}$ //ignorer un nœud sur une branche;
6. $M\{S_1, S_2\} \longrightarrow \{MS_1, MS_2\}$ //distributivité à gauche;
7. $S \longrightarrow S \cup S'$, où $S \longrightarrow S'$ //ajouter des nouveaux termes–frères;
8. $S \cup S_1 \longrightarrow S' \cup S_1$, si $S \longrightarrow S'$ //réécrire certains termes–frères;
9. $\Downarrow \omega S \longrightarrow (\downarrow \omega S) \vee (\downarrow * \Downarrow \omega S)$
   //cas d'analyse: tout descendant est soit un enfant, soit sa profondeur est ≥ 2,
10. $\Downarrow \omega S \longrightarrow (\downarrow \omega S) \vee (\Downarrow * \downarrow \omega S)$ //idem.

On appelle *context–pattern* tout pattern ayant un symbole supplémentaire $\Diamond$, dit *trou*, à la place d'un de ses sous–termes non–racinés. Par exemple,

$$\mathcal{C} = f\{\downarrow a, \Downarrow b\{\Diamond, \downarrow d\}, \Downarrow *\}$$

est un context–pattern. Considérons un context–pattern $\mathcal{C}$ et un terme non–raciné X. On appelle *fill–in* de $\mathcal{C}$ avec X le pattern, noté $\mathcal{C}\blacklozenge X$, obtenu à partir du context–pattern $\mathcal{C}$ après avoir remplacé le symbole de trou par le terme non–raciné X. Par exemple, pour le context–pattern $\mathcal{C}$ donné plus haut, et le terme non–raciné $X = \Downarrow x\{\downarrow y, \Downarrow z\}$, on a

$$\mathcal{C}\blacklozenge X = f\{\downarrow a, \Downarrow b\{\Downarrow x\{\downarrow y, \Downarrow z\}, \downarrow d\}, \Downarrow *\}.$$

On suppose aussi que pour tout context–pattern $\mathcal{C}$, et des termes non–racinés X, X', l'expression $\mathcal{C}\blacklozenge(X \vee X')$ est une notation abrégée du d–pattern $\mathcal{C}\blacklozenge X \vee \mathcal{C}\blacklozenge X'$. Pour réécrire les termes avec des règles de $\mathcal{R}$, on utilisera la *réécriture suffixe* :

Définition 5.4. *Soient un pattern* P *et un pattern ou un d–pattern* Q. *On dit que* P *peut être réécrit en un pas vers* Q *en utilisant la réécriture suffixe (on note* $P \longrightarrow_{\mathcal{R}} Q$*), s'il exsite un context–pattern* $\mathcal{C}$ *et deux termes non–racinés* X *et* X', *tels que :* $P = \mathcal{C}\blacklozenge X$, $Q = \mathcal{C}\blacklozenge X'$, *et* $X \longrightarrow X'$ *est une instance d'une des règles du système* $\mathcal{R}$.

De plus, les termes disjonctifs peuvent être réécrits en utilisant deux règles supplémentaires suivantes, où P est un pattern, et D, D_1, D_2 sont des d–patterns :

11. $D_1 \vee D \longrightarrow D_2 \vee D$, si $D_1 \longrightarrow D_2$ //réécriture des cas d'analyse;
12. $P \vee P \vee D \longrightarrow P \vee D$ //considérer tout cas d'analyse une seule fois.

Le résultat principal de ce chapitre est le suivant :

Théorème 5.1. *Soient P et Q deux patterns du fragment XP(/,//,[ ],*), on a :*

$$P \subseteq Q \Longleftrightarrow P \xrightarrow{*}_{\mathcal{R}} Q,$$

c.–à–d., $P \subseteq Q$ si et seulement si, on peut réécrire P vers Q en utilisant les règles de $\mathcal{R}$.

Preuve. Soient deux patterns P et Q sur un alphabet Σ. La sémantique des règles dans $\mathcal{R}$ garantit que $P \xrightarrow{*}_{\mathcal{R}} Q$ implique $P \subseteq Q$. En effet, si $X \longrightarrow X'$ est une instance d'une des règles 1−10, alors pour tout context–pattern $\mathcal{C}$, on a $\mathcal{C}\blacklozenge X \subseteq \mathcal{C}\blacklozenge X'$; de même, si $L \longrightarrow R$ est une instance de la règle 11 ou 12, on a bien évidemment $L \subseteq R$.

Pour montrer la réciproque, on commence par le lemme suivant :

Lemme 5.1. *Soient deux patterns P et Q. S'il existe un homomorphisme de Q vers P, alors $P \xrightarrow{*}_{\mathcal{R}} Q$.*

Preuve. Soit φ un homomorphisme donné de Q vers P. En utilisant φ, on construira un pattern P', tel que $P \xrightarrow{*}_{\mathcal{R}} P' \xrightarrow{*}_{\mathcal{R}} Q$:
(a) pour chaque nœud u de Q, on construit un nœud correspondant u' sur P', en posant $name_{P'}(u') = name_P(\varphi(u))$,
(b) on construit une arête simple $(u', v') \in Edges_{\downarrow}(P')$, si et seulement si $(u, v) \in Edges_{\downarrow}(Q)$
(c) on construit une arête double $(u', v') \in Edges_{\Downarrow}(P')$, si et seulement si $(u, v) \in Edges_{\Downarrow}(Q)$.
Le coût de la construction du pattern P' est linéaire par rapport au nombre d'arêtes de Q. Il est évident que le pattern P' ainsi construit se réécrit vers le pattern Q, en utilisant la règle 3. En effet, remarquons que P' a la même structure que Q. Les seules différences entre P' et Q peuvent être des noms de nœuds correspondant $u \in Nodes_Q$ et $u' \in Nodes_{P'}$ (condition (a)), car : soit $name_Q(u) = name_{P'}(u') = name_P(\varphi(u))$, soit $name_Q(u) \neq name_{P'}(u') = name_P(\varphi(u))$. La définition d'homomorphisme implique que dans le deuxième cas on a : $name_Q(u) = *$ et $name_{P'}(u') \in \Sigma$ (dans ce cas, en réécrivant P' vers Q on utilisera la règle 3). Il reste à montrer qu'on peut réécrire P vers P'. Dans un premier temps, en utilisant les règles 1 et 8 (où $S' = \emptyset$), on peut ignorer des sous–branches de P qui ne contiennent pas de nœuds images par φ. Ensuite, si un nœud w de P est l'image par φ de m nœuds distincts de Q, on réécrit P vers P' en construisant m nœuds distincts de P' (à partir d'un seul nœud de P) en utilisant la règle 7 (pour $S' = S$) et/ou la règle 6. Considérons maintenant l'arête $(u', v') \in Edges_{\downarrow}(P')$. Grâce à la condition (b) on sait que $(u, v) \in Edges_{\downarrow}(Q)$, d'où par la définition d'un homomorphisme, on a aussi $(\varphi(u), \varphi(v)) \in Edges_{\downarrow}(P)$ (rien à faire en réécrivant P vers P'). Soit maintenant l'arête $(u', v') \in Edges_{\Downarrow}(P')$. La condition (c) et la définition d'homomorphisme impliquent qu'il existe $k \geq 1$ et $w_0, \ldots w_k \in Nodes_P$, tels que : $w_0 = \varphi(u), w_k = \varphi(v)$, et $\forall\ i \in \{0, \ldots, k-1\}$ on a

$(w_i, w_{i+1}) \in Edges_{\downarrow}(P) \cup Edges_{\Downarrow}(P)$. Si $k = 1$ et $(\varphi(u), \varphi(v)) \in Edges_{\downarrow}(P)$, alors on réécrit P vers P' en utilisant le règle 4. Si $k \geq 2$, on utilise la règle 5 pour se débarrasser des nœuds $w_1, \ldots w_{k-1}$ en réécrivant P vers P'. Finalement on obtient $P \xrightarrow{*}_{\mathcal{R}} P' \xrightarrow{*}_{\mathcal{R}} Q$.

□

Notons que dans le cas où P est un arbre on a également la réciproque du lemme précédent, c.–à–d., si on peut réécrire un arbre P vers un pattern Q, alors il existe un homomorphisme de Q vers P (voir [50]). Bien évidemment, cet homomorphisme est un plongement du pattern Q dans l'arbre P, d'où P est un modèle de Q. On a alors :

Remarque 5.1. *Un arbre t est modèle d'un pattern P ssi*

Le système de réécriture $\mathcal{R}$ est non–déterministe. Néanmoins, si P et Q sont donnés, on peut définir une stratégie *dirigée par but* ("goal–directed") qui permet de réécrire P vers Q d'une façon optimale : l'idée est de réécrire P en utilisant *seulement* ces règles qui permettent de converger vers le pattern Q. On illustre ce concept dans l'exemple ci–dessous, où on reprend les patterns P et Q de la Figure 5.5. On sait que $P \subseteq Q$, mais il n'existe pas d'homomorphisme de Q vers P. On montre ici que $P \xrightarrow{*}_{\mathcal{R}} Q$.

Exemple 5.3. *Considérons l'arête soulignée $c \Downarrow d$ du pattern P ci–dessous :*

$$P = a \Downarrow b\{\downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow \underline{c \Downarrow d}\}, \downarrow c \downarrow * \Downarrow d\}$$

*Les modèles de P peuvent réaliser cette arête de deux façons : soit par une arête simple (*child*) $c \downarrow d$, soit par une chaîne de longueur ≥ 2 d'arêtes simples $c \downarrow \ldots \downarrow d$. On remarque que le pattern P peut être vu comme le fill–in*

$$a \Downarrow b\{\downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c\Diamond\}, \downarrow c \downarrow * \Downarrow d\} \ \blacklozenge \ \Downarrow d,$$

d'où, en utilisant le règle 9 *du système $\mathcal{R}$ on obtient le d–pattern qui représente les cas d'analyse considérés dans l'exemple 5.1 :*

$$\begin{array}{ll} & a \Downarrow b\{\downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c\Diamond\}, \downarrow c \downarrow * \Downarrow d\} \ \blacklozenge \ \downarrow d \\ \vee & a \Downarrow b\{\downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c\Diamond\}, \downarrow c \downarrow * \Downarrow d\} \ \blacklozenge \ \downarrow * \Downarrow d. \end{array}$$

Ce d–pattern n'est autre que

$$\begin{array}{ll} & a \Downarrow b\{\downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow d\}, \downarrow c \downarrow * \Downarrow d\} \\ \vee & a \Downarrow b\{\downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow * \Downarrow d\}, \downarrow c \downarrow * \Downarrow d\}, \end{array}$$

et il peut être vu sous la forme suivante :

$$\begin{array}{ll} & a \Downarrow b\{\downarrow b\{\Diamond, \downarrow c \downarrow d\}, \downarrow c \downarrow * \Downarrow d\} \ \blacklozenge \ \underline{\downarrow b \downarrow c \downarrow d} \\ \vee & a \Downarrow b\{\downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow * \Downarrow d\}, \Diamond\} \ \blacklozenge \ \underline{\downarrow c \downarrow * \Downarrow d}. \end{array}$$

En utilisant la règle 11 *on réécrit séparément chaque pattern qui compose le d–pattern disjonctif obtenu. On utilise ici la règle* 1 *pour ignorer tout simplement les branches soulignées :*

$$a \Downarrow b\{\downarrow b\{\downarrow c \downarrow d\}, \downarrow c \downarrow * \Downarrow d\} \quad \vee \quad a \Downarrow b\{\downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow * \Downarrow d\}\}.$$

Grâce à la règle 2*, on peut identifier le term obtenu avec le d–pattern :*

$$a \Downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow * \Downarrow d\} \quad \vee \quad a \Downarrow b \downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow * \Downarrow d\}.$$

Sa première composante est égale au pattern Q, et la deuxième peut être vue comme le fill–in suivant :

$$a\Diamond \ \blacklozenge \ \underline{\Downarrow b \downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow * \Downarrow d\}}.$$

La règle 5 *nous permet d'ignorer le premier nœud b dans le terme souligné, ce qui nous donne :*

$$a\Diamond \blacklozenge \Downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow * \Downarrow d\} = a \Downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow * \Downarrow d\}.$$

En résumant, on obtient le d–pattern

$$a \Downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow * \Downarrow d\} \quad \vee \quad a \Downarrow b\{\downarrow b \downarrow c \downarrow d, \downarrow c \downarrow * \Downarrow d\} = Q \vee Q,$$

qui est finalement réécrit vers le pattern Q en utilisant la règle 12*. En employant le Théorème 5.1 on a alors* $P \subseteq Q$*.*

Quelques remarques

Notons que l'approche de ce chapitre présente le travail en cours, et que la formalisation (à l'aide d'un algorithme) de la stratégie dirigée par but, illustrée dans l'Exemple 5.3, fait partie des travaux envisagés dans l'avenir.

L'Exemple 5.4 permet de remarquer que, notre approche pour l'inclusion de patterns n'est pas valide si on n'utilise pas la réécriture du type suffixe.

Exemple 5.4. *Considérons les patterns* $P = * \Downarrow *$*, et* $Q = * \downarrow *$*. On a bien évidemment* $P \subseteq Q$ *(*$P \xrightarrow{*}_{\mathcal{R}} Q$ *en utilisant les règles* 10 *et* 1*). Néanmoins,*

$$P\Diamond \blacklozenge \downarrow a = * \Downarrow * \downarrow a$$

n'est plus inclus dans

$$Q\Diamond \blacklozenge \downarrow a = * \downarrow * \downarrow a,$$

car par exemple, l'arbre $t = f \downarrow g \downarrow b \downarrow a$ *est un modèle pour* $* \Downarrow * \downarrow a$*, mais n'est pas un modèle pour* $* \downarrow * \downarrow a$*.*

Dans [32] nous montrons que l'approche présentée dans cette section reste valide même lorsque les modèles des patterns sont donnés sous une forme compressée. Nous y esquissons également comment adapter cette approche en vue d'obtenir une méthode d'évaluation de requêtes, unaires ainsi que n–aires, du fragment XP(/,//,[ ],*).

Conclusion

Dans ce travail nous nous sommes intéressés aux requêtes XPath, et leur évaluation sur les documents XML arborescents ou compressés, et assujettis à des politiques du contrôle d'accès. Notre objectif principal était de concevoir des approches :

- pouvant utiliser directement les documents d'arité non fixe (unranked) et non pas vus à travers leurs représentations binaires,
- pouvant traiter les documents compressés sans nécessiter une décompression.

Selon le contexte, nous avons employé : les automates, les systèmes de transitions, et la réécriture. Les résultats obtenus montrent que ces techniques classiques se prêtent bien au traitement des requêtes sur les documents XML, selon un tel "cahier de charges". Dans le cas où on se restreint au fragment purement descendant de XPath, toutes les approches introduites dans ce travail s'adaptent, sans problème, aux documents compressés représentés à l'aide de dags.

Nous avons introduit deux mécanismes permettant d'évaluer des requêtes XPath

1. sur les documents XML compressés,
2. en présence d'une politique du contrôle d'accès.

Le fait d'utiliser les documents XML modélisés par des arbres ou des dags d'arité non fixe ni bornée explique pourquoi nous avons choisi les automates de mots et les systèmes de transitions, plutôt que les automates d'arbres ou de dags. La complexité en temps de nos approches est du même ordre que la meilleure complexité connue pour les problèmes abordés. Dans ce travail nous n'avons considéré que des requêtes positives, c.–à–d., sans négation dans leur partie de navigation. Nous étudions actuellement les adaptations nécessaires pour étendre l'approche du Chapitre 3 aux requêtes a0]TJ36a480d199exp990Td3381. n n14.03980Tdégaes

Bibliographie

[1] Martin Abadi. Logic in Access Control. In *Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03)*, pages 228–233, Ottawa, Canada, June 2003. IEEE Computer Society Press. 15.

[2] Martín Abadi and Bogdan Warinschi. Security Analysis of Cryptographically Controlled Access to XML Documents. In *PODS '05: Proceedings of the twenty-fourth ACM SI MOD-SI ACT-SI ART symposium on Principles of database systems*, pages 108–117, New York, NY, USA, 2005. ACM.

[3] Joaquin Adiego, Pablo De la Fuente, and Gonzalo Navarro. Combining Structural and Textual Contexts for Compressing Semistructured Databases. In *ENC '05: Proceedings of the Sixth Mexican International Conference on Computer Science*, pages 68–73, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of Tree Pattern Queries. In *SI MOD '01: Proceedings of the 2001 ACM SI MOD International Conference on Management of Data*, pages 497–508, New York, NY, USA, 2001. ACM.

[5] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Tree Pattern Query Minimization. *VLDB J.*, 11(4):315–331, 2002.

[6] Amihood Amir, Gary Benson, and Martin Farach. Let Sleeping Files Lie: Pattern Matching in Z–compressed Files. *J. Comput. Syst. Sci.*, 52(2):299–307, 1996.

[7] Siva Anantharaman, Paliath Narendran, and Michael Rusinowitch. Closure Properties and Decision Problems of DAG Automata. *Inf. Process. Lett.*, 94(5):231–240, 2005.

[8] Andrei Arion, Angela Bonifati, Gianni Costa, Sandra D'Aguanno, Ioana Manolescu, and Andrea Pugliese. Efficient Query Evaluation over Compressed Xml Data. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *EDBT*, volume 2992 of *Lecture Notes in Computer Science*, pages 200–218. Springer, 2004.

[9] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural Properties of XPath Fragments. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 79–95, London, UK, 2002. Springer-Verlag.

[10] Michael Benedikt and Christoph Koch. XPath Leashed. Available on: `http://web.comlab.ox.ac.uk/people/Michael.Benedikt/pub.html`. To Appear in ACM Computing Surveys, March 2009.

[11] Véronique Benzaken, Marwan Burelle, and Giuseppe Castagna. Information Flow Security for XML Transformations. In *ASIAN'03*, Lectures Notes in Computer Science. Spinger Verlag, 2003.

[12] Piotr Berman, Marek Karpinski, Lawrence L. Larmore, Wojciech Plandowski, and Wojciech Rytter. The complexity of Two–Dimensional Compressed Pattern–Matching. Technical report, 1996.

[13] Elisa Bertino, Silvana Castano, Elena Ferrari, and Marco Mesiti. Protection and administration of xml data sources. *Data Knowl. Eng.*, 43(3):237–260, 2002.

[14] Elisa Bertino and Elena Ferrari. Secure and Selective Dissemination of XML Documents. *ACM Trans. Inf. Syst. Secur.*, 5(3):290–331, 2002.

[15] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A Formal Model for an Expressive Fragment of XSLT. *Inf. Syst.*, 27(1):21–39, 2002.

[16] Peter Buneman, Martin Grohe, and Christoph Koch. Path Queries on Compressed XML. In *Very Large Databases (VLDB 2003), Berlin ermany*, page 12. Morgan Kaufmann, 2003.

[17] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Grammar–Based Tree Compression. Technical report, 2004.

[18] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient Memory Representation of XML Document Trees. *Inf. Syst.*, 33(4-5):456–474, 2008.

[19] Julien Carme, Joachim Niehren, and Marc Tommasi. Querying Unranked Trees with Stepwise Tree Automata. In Vincent van Oostrom, editor, *International Conference on Rewriting Techniques and Applications, Aachen*, volume 3091 of *Lecture Notes in Computer Science*, pages 105–118. Springer, June 2004.

[20] Silvana Castano, Maria Grazia Fugini, Giancarlo Martella, and Pierangela Samarati. *Database Security*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[21] Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Databases. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, New York, NY, USA, 1977. ACM.

[22] Witold Charatonik. Automata on DAG Representations of Finite Trees.

[23] James Cheney. Compressing XML with Multiplexed Hierarchical PPM Models. In *DCC '01: Proceedings of the Data Compression Conference (DCC '01)*, page 163, Washington, DC, USA, 2001. IEEE Computer Society.

[24] James Cheney. An empirical evaluation of simple dtd-conscious compression techniques. In AnHai Doan, Frank Neven, Robert McCann, and Geert Jan Bex, editors, *WebDB*, pages 43–48, 2005.

[25] 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.

[26] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Securing XML Documents. *Lecture Notes in Computer Science*, 1777:121–136, 2000.

[27] 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.

[28] Wenfei Fan, Chee-Yong Chan, and Minos Garofalakis. Secure XML Querying with Security Views. In *SI MOD '04: Proceedings of the 2004 ACM SI MOD international conference on Management of data*, pages 587–598, New York, NY, USA, 2004. ACM.

[29] Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *J. ACM*, 52(4):552–581, 2005.

[30] Barbara Fila and Siva Anantharaman. Automata for Positive Core XPath Queries on Compressed Documents. In *LPAR'06: In Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, LNAI 4246, pages 467–481. Springer-Verlag, 2006.

[31] Barbara Fila and Siva Anantharaman. A Clausal View for Access Control and XPath Query Evaluation (extended abstract). In *Pre–proceedings of The 17th International Symposium on Logic–Based Program Synthesis and Transformation (LOPSTR'07)*, pages 178–186. The Technical University of Denmark, 2007.

[32] Barbara Fila-Kordy. A Rewrite Approach for Pattern Containment – Application to Query Evaluation on Compressed Documents. In Stefan Böttcher, Markus Lohrey, Sebastian Maneth, and Wojciech Rytter, editors, *Structure-Based Compression of Complex Massive Data*, number 08261 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. Available on: `http://drops.dagstuhl.de/opus/volltexte/2008/1679/pdf/08261.Fila_KordyBarbara.Paper.1679.pdf`.

[33] Sergio Flesca, Filippo Furfaro, and Elio Masciari. On the Minimization of XPath Queries. *J. ACM*, 55(1), 2008.

[34] Markus Frick and Martin Grohe. The Complexity of First–order and Monadic Second–Order Logic Revisited. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 215–224, Washington, DC, USA, 2002. IEEE Computer Society.

[35] Markus Frick, Martin Grohe, and Christoph Koch. Query Evaluation on Compressed Trees (Extended Abstract). In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 188, Washington, DC, USA, 2003. IEEE Computer Society.

[36] Irini Fundulaki and Maarten Marx. Specifying Access Control Policies for XML Documents with XPath. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 61–69, New York, NY, USA, 2004. ACM.

[37] Alban Gabillon and Emmanuel Bruno. Regulating Access to XML Documents. In *Das'01: Proceedings of the fifteenth annual working conference on Database and application security*, pages 299–314, Norwell, MA, USA, 2002. Kluwer Academic Publishers.

[38] Leszek Gasieniec, Marek Karpiński, Wojciech Plandowski, and Wojciech Rytter. Efficient Algorithms for Lempel–Ziv Encoding (Extended Abstract). In *SWAT '96: Proceedings of the 5th Scandinavian Workshop on Algorithm Theory*, pages 392–403, London, UK, 1996. Springer-Verlag.

[39] Georg Gottlob and Christoph Koch. Monadic Datalog and the Expressive Power of Languages for Web Information Extraction. In *Symposium on Principles of Database Systems (PODS)*, pages 17–28, 2002.

[40] Georg Gottlob and Christoph Koch. Monadic Queries over Tree–Structured Data. In *Proc. LICS*, Copenhagen, 2002.

[41] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 95–106. VLDB Endowment, 2002.

[42] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The Complexity of XPath Query Evaluation. In *PODS '03: Proceedings of the twenty-second ACM SI MOD-SI ACT-SI ART symposium on Principles of database systems*, pages 179–190, New York, NY, USA, 2003. ACM.

[43] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. The Complexity of XPath Query Evaluation and XML Typing. *J. ACM*, 52(2):284–335, 2005.

[44] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. Technical report, University of Washington, 2001.

[45] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997.

[46] Pieter H. Hartel. A Trace Semantics for Positive Core XPath. In *TIME '05: Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 103–112, Washington, DC, USA, 2005. IEEE Computer Society.

[47] Juhani Karhumäki, Wojciech Plandowski, and Wojciech Rytter. The Complexity of Compressing Subsegments of Images Described by Finite Automata. *Discrete Appl. Math.*, 125(2-3):235–254, 2003.

[48] Benny Kimelfeld and Yehoshua Sagiv. Revisiting Redundancy and Minimization in an XPath Fragment. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 61–72, New York, NY, USA, 2008. ACM.

[49] Anne Klein, Makoto Murata, and Derick Wood. Regular Tree and Regular Hedge Languages over Unranked Alphabets. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[50] Barbara Kordy. A Rewrite Approach for Pattern Containment. In Andrea Corradini and Ugo Montanari, editors, *WADT'08: Recent Trends in Algebraic Development Techniques*, LNCS. Springer-Verlag, 2009.

[51] Tokyo Research Lab. XML Access Control Project. Available on: `http://www.trl.ibm.com/projects/xml/xacl/`.

[52] Gregory Leighton, Jim Diamond, and Tomasz Müldner. AXECHOP: A Grammar-based Compressor for XML. In *DCC '05: Proceedings of the Data Compression Conference*, pages 467–467, Washington, DC, USA, 2005. IEEE Computer Society.

[53] Gregory Leighton, Tomasz Müldner, and James Diamond. TREECHOP: A Tree-based Query–able Compressor for XML. Available on: `http://cs.acadiau.ca/research/technicalReports/files/tr-2005-005.pdf`, 2005. TR–2005–005, Acadia University.

[54] Weimin Li. XComp: An XML Compression Tool. Available on: `http://softbase.uwaterloo.ca/~ddbms/publications/distdb/Weimin.pdf`, 2003. Master's thesis, University of Waterloo.

[55] Hartmut Liefke and Dan Suciu. XMill: an Efficient Compressor for XML Data. In *SI MOD '00: Proceedings of the 2000 ACM SI MOD international conference on Management of data*, pages 153–164, New York, NY, USA, 2000. ACM.

[56] Yury Lifshits and Markus Lohrey. Querying and Embedding Compressed Texts. In *MFCS*, pages 681–692, 2006.

[57] Chung-Hwan Lim, Seog Park, and Sang H. Son. Access Control of XML Documents Considering Update Operations. In *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, pages 49–59, New York, NY, USA, 2003. ACM.

[58] Markus Lohrey. Word Problems and Membership Problems on Compressed Words. *SIAM J. Comput.*, 35(5):1210–1240, 2006.

[59] Markus Lohrey and Sebastian Maneth. Tree Automata and XPath on Compressed Trees. In *CIAA*, pages 225–237, 2005.

[60] Markus Lohrey and Sebastian Maneth. The Complexity of Tree Automata and XPath on Grammar–compressed Trees. *Theoretical Computer Science*, 363(2):196–210, 2006.

[61] Sebastian Maneth and Giorgio Busatto. Tree Transducers and Tree Compressions. In Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2004.

[62] Wim Martens and Joachim Niehren. On the Minimization of XML Schemas and Tree Automata for Unranked Trees. *J. Comput. Syst. Sci.*, 73(4):550–583, 2007.

[63] Maarten Marx. XPath with Conditional Axis Relations. In EDBT, pages 477–494, Crete, Greece, March 2004.

[64] Maarten Marx. XPath and Modal Logics of Finite DAG's. In *TABLEAUX*, pages 150–164, 2003.

[65] Maarten Marx. Conditional XPath. *ACM Trans. Database Syst.*, 30(4):929–959, 2005.

[66] Gerome Miklau and Dan Suciu. Controlling Access to Published Data Using Cryptography. In *vldb'2003: Proceedings of The 29th International Conference on Very Large Data Bases*, pages 898–909. VLDB Endowment, 2003.

[67] Gerome Miklau and Dan Suciu. Containment and Equivalence for a Fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[68] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *ICDT '99: Proceedings of the 7th International Conference on Database Theory*, pages 277–295, London, UK, 1999. Springer-Verlag.

[69] Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. XPRESS: a Queriable Compression for XML Data. In *SI MOD '03: Proceedings of the 2003 ACM SI MOD international conference on Management of data*, pages 122–133, New York, NY, USA, 2003. ACM.

[70] Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda. An Improved Pattern Matching Algorithm for Strings in Terms of Straight–Line Programs. In *CPM '97: Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, pages 1–11, London, UK, 1997. Springer-Verlag.

[71] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. XML Access Control Using Static Analysis. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 73–84, New York, NY, USA, 2003. ACM.

[72] Andreas Neumann and Helmut Seidl. Locating Matches of Tree Patterns in Forests. In *Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science*, pages 134–145. Springer, 1998.

[73] Frank Neven. Automata, Logic, and XML. In *CSL '02: Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic*, pages 2–26, London, UK, 2002. Springer-Verlag.

[74] Frank Neven. Automata Theory for XML Researchers. *SI MOD Rec.*, 31(3):39–46, 2002.

[75] Frank Neven and Thomas Schwentick. Expressive and Efficient Pattern Languages for Tree–Structured Data. In *Proceedings of the Nineteenth ACM SI MOD-SI ACT-SI ART Symposium on Principles of Database Systems*, pages 145–156. ACM, 2000.

[76] Frank Neven and Thomas Schwentick. Query Automata over Finite Trees. *Theor. Comput. Sci.*, 275(1-2):633–674, 2002.

[77] Frank Neven and Thomas Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, volume 2572 of *LNCS*, pages 312–326. Springer, 2003.

[78] Joachim Niehren, Laurent Planque, Jean-Marc Talbot, and Sophie Tison. N–ary Queries by Tree Automata. In *10th International Symposium on Database Programming Languages*, September 2005.

[79] OASIS. eXtensible Access Control Markup Language (XACML). Available on: `http://www.oasis-open.org/specs/`. approved as OASIS Standards on 1 February 2005.

[80] Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In *ESA '94: Proceedings of the Second Annual European Symposium on Algorithms*, pages 460–470, London, UK, 1994. Springer-Verlag.

[81] Wojciech Plandowski and Wojciech Rytter. Complexity of Language Recognition Problems for Compressed Words. In *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 262–272, London, UK, 1999. Springer-Verlag.

[82] Prakash Ramanan. Efficient Algorithms for Minimizing Tree Pattern Queries. In *SI MOD '02: Proceedings of the 2002 ACM SI MOD International Conference on Management of Data*, pages 299–309, New York, NY, USA, 2002. ACM.

[83] Wojciech Rytter. Algorithms on Compressed Strings and Arrays. In *SOFSEM '99: Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*, pages 48–65, London, UK, 1999. Springer-Verlag.

[84] Wojciech Rytter. Application of Lempel–Ziv Factorization to the Approximation of Grammar–based Compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[85] Thomas Schwentick. XPath Query Containment. *SI MOD Rec.*, 33(1):101–109, 2004.

[86] Przemyslaw Skibiński, Szymon Grabowski, and Jakub Swacha. Effective Asymmetric XML Compression. Available on: `http://www.ii.uni.wroc.pl/ inikep/papers /07-AsymmetricXML.pdf`, 2007. To appear in: *Software: Practice and Experience, 2008.*

[87] William Stallings. *Network Security Essentials: Applications and Standards.* Prentice Hall Professional Technical Reference, 2002.

[88] Pankaj M. Tolani and Jayant R. Haritsa. XGRIND: A Query–Friendly XML Compressor. *ICDE*, 00:0225, 2002.

[89] Peter T. Wood. On the Equivalence of XML Patterns. In *CL '00: Proceedings of the First International Conference on Computational Logic*, pages 1152–1166, London, UK, 2000. Springer-Verlag.

[90] Peter T. Wood. Minimising Simple XPath Expressions. In *WebDB*, pages 13–18, 2001.

[91] World Wide Web Consortium. Overview of SGML Resources. Available on: `http://www.w3.org/MarkUp/SGML/`, 1995. Created November.

[92] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. Available on: `http://www.w3.org/TR/xpath`, 1999. W3C Recommendation 16 November 1999.

[93] World Wide Web Consortium. XSL Transformations (XSLT) version 1.0. Available on: `http://www.w3.org/TR/xslt`, 1999. W3C Recommendation 16 November.

[94] World Wide Web Consortium. XML Linking Language (XLink) Version 1.0. Available on: `http://www.w3.org/TR/xlink/`, 2001. W3C Recommendation 27 June.

[95] World Wide Web Consortium. XML Schema. Available on: `http://www.w3.org/XML/Schema`, 2001. W3C Recommendation on 2 May.

[96] World Wide Web Consortium. XML Pointer Language (XPointer). Available on: `http://www.w3.org/TR/xptr/`, 2002. W3C Working Draft 16 August 2002.

[97] World Wide Web Consortium. Extensible Markup Language (XML). Available on: `http://www.w3.org/TR/REC-xml/`, 2006. W3C Recommendation 16 August 2006, edited in place 29 September.

[98] World Wide Web Consortium. XML Path Language (XPath) 2.0. Available on: `http://www.w3.org/TR/xpath20/`, 2007. W3C Recommendation 23 January.

[99] World Wide Web Consortium. XQuery 1.0: An XML Query Language. Available on: `http://www.w3.org/TR/xquery/`, 2007. W3C Recommendation 23 January.

[100] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 82–94. VLDB Endowment, 1981.

[101] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *j-IEEE-TRANS-INF-THEORY*, IT-23(3):337–343, May 1977.

[102] Jacob Ziv and Abraham Lempel. Compression of Individual Sequences via Variable Rate Coding. *j-IEEE-TRANS-INF-THEORY*, IT-24(5):530–536, September 1978.

Index