Une méthode pour l'évolution des schémas XML préservant la validité des documents

Denio Duarte

UNIVERSITÉ FRANÇOIS-RABELAIS - TOURS

LABORATOIRE D'INFORMATIQUE

École doctorale Santé, Sciences, Technologies

Année Universitaire : 2004-2005

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITÉ FRANÇOIS-RABELAIS - TOURS

Discipline : Informatique

présentée et soutenue publiquement par :

DENIO DUARTE

le 04 juillet 2005

TITRE

Une méthode pour l'évolution de schémas XML préservant la validité des documents

Directeur de thèse :

Dominique LAURENT

Co-encadré par :

Béatrice BOUCHOU
Mírian HALFELD FERRARI ALVES

JURY :

Benzaken Véronique	Professeur	Université de Paris-Sud XI
Bouchou Béatrice	Maître de conférences	Université François-Rabelais - Tours
Champarnaud Jean-Marc	Professeur	Université de Rouen
Halfeld Ferrari Alves Mírian	Maître de conférences	Université François-Rabelais - Tours
Laurent Dominique	Professeur	Université Cergy-Pontoise
Vrain Christel	Professeur	Université d'Orléans

Cette thèse a été financée par le gouvernement brésilien - CAPES (BEX0353/01-9)

...

Car enfin, qu'est-ce que l'homme dans la nature ? Un néant à l'égard de l'infini, un tout à l'égard du néant, un milieu entre rien et tout. Infiniment éloigné de comprendre les extrêmes, la fin des choses et leurs principes sont pour lui invinciblement cachés dans un secret impénétrable également incapable de voir le néant d'où il est tiré et l'infini où il est englouti.

Blaise Pascal, Les pensées (section II, Pensé 72).

E agora, José?
A festa acabou,
a luz apagou,
o povo sumiu,
a noite esfriou,
e agora, José?
e agora, você?
você que é sem nome,
que zomba dos outros,
você que faz versos,
que ama, protesta?
e agora, José?

...

José (1941) - Carlos Drummond de Andrade

Si j'ai réussi à avoir mon doctorat, c'est grâce à quatre femmes auxquelles je dédie cette thèse :

Dona Lydia, ma mère, une femme qui a toujours fait et qui fait encore le possible *et l'impossible* pour ses enfants.

Iara, ma sœur, qui après le décès de mon père a sacrifié sa jeunesse pour assumer une tâche trop lourde mais qu'elle a bien su tenir et par la force de l'habitude elle la fait encore.

Ana, mon épouse, mon *nord*. Pendant toutes ses années, elle a toujours été là, prête pour m'écouter, pour me faire plaisir. Toutes ces années auraient été difficiles sans elle.

Cecília, ma fille. Sa tendresse, son amour et ses petites questions me font toujours oublier les problèmes de tous les jours.

Remerciements

Je suis doublement content d'écrire cette partie : d'abord, le doctorat est déjà un pas franchi et ensuite, elle ne sera pas révisée par mes encadreurs (et donc, vous allez trouver des erreurs).

Il y a des nombreuses personnes à qui je voudrais remercier pour le soutien pendant les années de mon doctorat.

Tout d'abord je voudrais remercie très sincèrement les membres du jury :

Dominique Laurent de m'avoir accepté au sein du laboratoire d'informatique de l'Université François Rabelais de Tours (campus Blois) et de me diriger pendant ces presque quatre ans.

Béatrice Bouchou et Mírian Halfeld Ferrari Alves d'avoir accepté de me co-encadrer. Pendant des discussions que l'on a eues beaucoup d'idées sont nées. J'espère avoir appris à rédiger des textes scientifiques aussi bien qu'elles. Un merci spécial à Béatrice pour avoir compris si vite ce que j'ai expliqué si mal.

Véronique Benzaken et Jean-Marc Champarnaud m'ont fait l'honneur d'être rapporteurs de cette thèse. Je leur suis très reconnaissant d'avoir accepté cette lourde tâche et de l'intérêt qu'ils ont porté à mon travail.

Christel Vrain m'a fait l'honneur d'accepter de présider ce jury.

Je voudrais remercier aussi Martin Musicante. Les discussions pendant son séjour au notre laboratoire m'a beaucoup aidé dans la recherche des solutions pour l'évolution de schémas.

Je tiens à remercier le gouvernement brésilien (CAPES) de m'avoir donné l'opportunité de faire mon doctorat en France. Grâce à cette bourse, j'ai pu faire mon doctorat de façon tranquille, sans avoir trop de soucis par rapport à l'argent.

Je voudrais remercier toute l'équipe de l'Antenne Universitaire de Blois pour l'accueil très chaleureux et leurs disponibilités lorsque j'avais besoin d'aide. Mes collègues de bureau méritent aussi un grand merci de m'avoir supporté, soutenu, encouragé, motivé, embêté. Je cite d'abord ceux qui ont été toujours là, Adriana et Ahmed. Adriana est une personne que j'admire beaucoup par sa force et sa gentillesse. Ahmed, on a commencé presque ensemble avec ses histoires des automates et je suis très content de l'avoir croisé sur mon chemin. J'attends maintenant avec impatience que les deux aient eux aussi la joie de rédiger, à leur tour, leurs propres remerciements. Les nouveaux arrivants : Tonio et Cheikh Bah. Deux mois à peine, on est devenu une famille, merci Cheikh pour les astuces de français et Tonio pour les autres astuces. Malgré la thèse, on s'est beaucoup amusé. Je voudrais aussi remercier Cheikh Diop (le premier docteur de notre campus) qui m'a filé pas mal d'astuces pour la thèse et Oumar un mec très sympa qui faisait partie de notre bande au début mais qu'a décidé de devenir parisien (la classe).

Maintenant, il faut traverser l'Atlantique. Merci à mon beau-frère, João, pour le soutien. Il (avec ma sœur) ne mesurait pas d'efforts pour nous aider, de même pour mes beaux-parents, *Seu* Jairo et *Dona* Vilma, pour des mots de soutien et d'encouragement dans les moments qu'il fallait. Je ne peux pas oublier ma petite belle-sœur, Carolina ou Carol ou Tataka, qui a passé 6 mois en France et qui m'a préparé pas mal de repas (parfois salé, parfois sans sel) très délicieux. Je voudrais aussi remercier *Monsieur* Barbosa (et sa famille), un ami auquel que je tiens beaucoup.

Table des matières

Table des figures

Chapitre 1

Introduction

Depuis sa première publication, en 1998, le langage XML, *eXtensible Markup Language*, est devenu un format de balisage des données utilisé par de nombreuses applications. Cela s'explique par le fait que XML est capable de baliser aussi bien un contenu *web* que des données utilisées par les applications. En fait, un document XML transporte les données et les informations sur les données transportées. Même s'il a été conçu pour traiter les documents, XML a été raffiné comme un modèle de données semi-structurées et il modélise de plus en plus un nombre croissant d'applications :

- Gestion d'informations : XML est utilisé pour représenter les informations que les applications peuvent utiliser, gérer et/ou afficher plus facilement.
- Intégration de données : XML est utilisé comme une technologie capable d'intégrer des données provenant de différentes sources.
- Interopérabilité : XML est utilisé comme une technologie permettant de combiner les résultats de plusieurs applications.
- Échange de données : XML est utilisé comme un protocole facilitant l'échange de données entre plusieurs applications indépendamment de la plate-forme.

La représentation de la structure logique des documents XML s'appuie sur trois entités [Roi99] :

- Les éléments de base non décomposables qui constituent le contenu.
- Les éléments composites obtenus par composition d'éléments de base ou d'éléments composites ; les opérateurs de composition sont l'identité, l'agrégation, la liste et le choix.
- Les attributs qui peuvent être associés aux éléments pour leur adjoindre des informations sémantiques.

Ces principes permettent d'organiser les documents selon une structure hiérarchique d'éléments typés, décrite linéairement comme un emboîtement de marques (ou balises) encapsulant les éléments de base. Des modèles génériques de description de structures logiques de documents permettent de décrire des classes pour les documents qui partagent les mêmes spécifications. On parle alors de schémas, qui peuvent être vus comme des grammaires.

Les documents XML associés à un schéma sont dits valides quand ils respectent toutes les contraintes (structurelles) imposées par le schéma. La figure 1.1 illustre le rôle des schémas dans XML : avec un schéma on peut modéliser un ensemble précis de documents.

Les applications qui s'appuient sur les schémas pour connaître les structures des données XML peuvent alors avoir des informations importantes pour optimiser les évaluations de requêtes, obtenir des informations générales sur le contenu, faciliter l'intégration de données issues de différentes sources d'information, améliorer le stockage, faciliter la mise en place d'index ou de vues et aider à la classification de documents [ABS00].

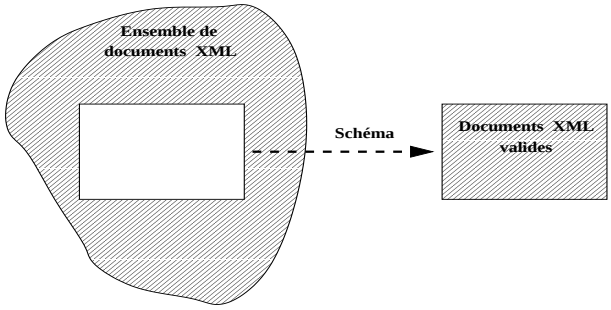


FIG. 1.1 – Le rôle d'un schéma dans un ensemble de documents XML.

Les documents XML peuvent évoluer : des nouveaux éléments sont ajoutés, des valeurs textuelles sont mises à jour, entre autres. Si les documents sont valides, le contrôle de leur évolution est fait par les schémas : un document qui appartient à une classe de documents doit toujours respecter les contraintes imposées par cette classe et les mises à jour doivent aussi respecter ces contraintes. Néanmoins, les schémas, à leur tour, ont besoin d'évoluer aussi et dans ce cas, l'évolution d'un schéma peut changer la classe de documents qu'il décrit. Ainsi, la mise à jour des schémas XML n'est pas une tâche triviale car il faut prendre en considération tous les documents conformes à un schéma avant la mise à jour pour vérifier s'il faut les changer pour qu'ils soient valides par rapport au schéma mis à jour [AE03].

La façon naturelle de faire évoluer un schéma est de le changer selon le besoin d'utilisateur responsable de l'application. Un tel utilisateur doit donc maîtriser non seulement son domaine d'application mais aussi les conséquences des changements effectués sur un schéma vis à vis des documents (initialement valides) créés par d'autres utilisateurs. En d'autres mots, la mise à jour d'un schéma implique la revalidation de tous les documents auparavant valides (et d'éventuels "aller-retour" sur les changements effectués). L'exemple suivant illustre cette situation.

Exemple 1.0.1 Considérons un schéma d qui modélise des documents qui stockent des publications d'un groupe de chercheurs. Ce schéma modélise les publications de la façon suivante : les publications sont composées des articles de revues qui sont organisés par sujets et par années de publication. L'expression régulière[1] $Sujet\ (Annee\ Revue^{+})^{*}$ décrit d. Soit le document XML ci-dessous, valide par rapport à d.

```
<Publications>
   <Sujet> Automates </Sujet>
   <Annee> 1995 </Annee>
   <Revue> Theoretical Computer Science </Revue>
   <Revue> International Journal of Computer Mathematics </Revue>
   <Annee> 1996 </Annee>
   <Revue> Theoretical Computer Science </Revue>
</Publications>
```

Tous les laboratoires d'une même université utilisent le schéma d non seulement pour échanger des informations entre eux mais aussi pour leurs documentations internes. Supposons que l'un des laboratoires veuille ajouter à ses documents XML ses travaux publiés dans des conférences. L'utilisateur responsable de la documentation du laboratoire décide de changer d pour qu'il modélise aussi les conférences. Supposons que cet utilisateur est un bibliothécaire non expert en informatique. Il décide donc de changer l'expression d'origine (en s'inspirant de l'expression régu-

[1] La majorité des schémas XML utilisent des expression régulières [LC00].

lière d'origine) en $Sujet\ (Annee\ Revue^{+}\ Conference^{+})^{*}$. Dans ce cas, tous les documents de la base de données deviendront invalides par rapport au nouveau schéma car l'élément $Conference$ a été inséré comme obligatoire. □

Le but de cette thèse est proposer un système qui aide les utilisateurs responsables des applications mais qui ne sont pas forcément des experts en informatique. Nous proposons ainsi une approche où l'utilisateur donne au système ce qu'il souhaite comme nouveau document devant être accepté par le schéma. À partir de ce document, le système construit des schémas candidats qui d'une part préservent la validité de la base de documents et, d'autre part augmentent la classe de documents acceptée par le schéma.

Le document donné au système comme exemple à accepter par le nouveau schéma est construit par l'administrateur responsable en changeant l'un des documents valides par rapport au schéma d'origine (un document que l'administrateur connaît bien). Ainsi, en s'appuyant sur le document exemple, le système propose des nouveaux schémas que l'utilisateur peut donc choisir pour remplacer le schéma d'origine. Ces propositions de schémas sont construites de façon à :

1. Maintenir la cohérence de la base de donnés sans modifier les documents préexistants valides.
2. Ne pas accepter une nouvelle classe de documents trop générique par rapport à la classe modélisée par le schéma d'origine.
3. Ne pas créer un nouveau schéma de façon triviale : capable de décrire *seulement* les documents modélisés par le schéma d'origine et le nouveau document donné comme paramètre.

Les propositions sont organisées de façon à faciliter le choix de l'utilisateur, comme illustré par l'exemple suivant.

Exemple 1.0.2 Reprenons l'exemple 1.0.1. Supposons que l'utilisateur responsable décide d'activer le système d'aide à l'évolution de schémas avec le document suivant :

```
<Publications>
  <Sujet> Automates </Sujet>
  <Annee> 1995 </Annee>
  <Revue> Theoretical Computer Science </Revue>
  <Revue> International Journal of Computer Mathematics </Revue>
  <Annee> 1996 </Annee>
  <Revue> Theoretical Computer Science </Revue>
  <Conference> Mathematical Foundations of Computer Science </Conference>
</Publications>
```

Rappelons que l'expression régulière qui modélise la base de donnés est $Sujet(AnneeRevue^{+})^{*}$. Dans ce contexte, notre système propose (entre autres) les schémas suivants :

1. $Sujet\ (Annee\ (Revue\ Conference?)^{+})^{*}$
2. $Sujet\ (Annee\ (Revue|Conference)^{+})^{*}$
3. $Sujet\ (Annee\ Revue^{+}\ Conference^{*})^{*}$

Remarquer que toutes ces propositions de schéma maintiennent la validité des documents préexistants car les expressions régulières proposées décrivent des langages qui incluent le langage décrit par l'expression régulière d'origine. Le document exemple proposé par l'utilisateur est aussi valide par rapport à toutes les propositions de schémas. L'administrateur peut alors décider quel schéma adopter d'après sa connaissance sémantique de l'application. □

La motivation de notre contribution se généralise à plus grande échelle : imaginons, par exemple, un système distribué dont les données sont placées sur différents sites, il est important que, après la mise à jour d'un schéma, tous les documents sur tous les sites maintiennent leur validité pour que les applications continuent à s'exécuter normalement.

Par exemple, supposons qu'un schéma XML modélise les notices des médicaments d'une industrie pharmaceutique, que cette industrie ait des sites dans plusieurs pays et que les applications se servent des données modélisées par le schéma qui se trouve dans l'un des sites pour afficher des informations aux utilisateurs. Supposons aussi que dans un pays les lois ont changé et il faut avoir dans la notice des médicaments le danger pour les animaux familiers. Dans ce contexte, l'évolution du schéma peut être faite de façon incrémentale : le schéma aura les nouveaux éléments qui modélisent les nouvelles informations sur les dangers pour les animaux, cependant, tous les documents de tous les sites resteront valides par rapport au nouveau schéma. Ainsi, les applications des autres sites responsables pour afficher les informations sur les médicaments continueront à marcher normalement.

Une méthode qui s'appuie sur un document exemple pour étendre une classe de documents préexistante est très utile car l'utilisateur aura des modèles de schéma qui préservent la validité des documents pour remplacer le schéma d'origine. Ce document exemple peut être un document auparavant valide qui, après une mise à jour, ne respecte plus son schéma.

La principale contribution de ce mémoire est de proposer une approche pour aider des utilisateurs administrateurs dans la tâche de l'évolution des schémas XML. Cette approche est fondée sur l'adaptation du schéma à un document auparavant valide mais qui a été modifié par l'administrateur pour servir comme exemple de base pour le nouveau schéma.

Notre système d'aide à l'évolution de schémas a pour base une méthode de validation incrémentale des documents XML. Cette méthode de validation utilise les automates d'arbre et peut être résumée comme suit :

- Les contraintes sont imposées par un schéma XML vu comme un automate d'arbre $\mathcal{A}$.
- Les documents XML sont vus comme des arbres d'arité non-bornée (*i.e.*, des arbres dont on ne connaît pas le nombre de fils d'un nœud *a priori*) ayant deux types de nœuds.
- La vérification de la validité d'un document est faite en exécutant l'automate d'arbre sur l'arbre qui représente le document. Si l'automate accepte l'arbre, alors le document est conforme à son schéma.

En mettant à jour un arbre accepté par $\mathcal{A}$, l'utilisateur administrateur construit un nouvel arbre qui serait invalide par rapport à $\mathcal{A}$. Néanmoins, dans notre contexte, cet arbre est utilisé pour guider la construction d'un nouvel automate d'arbre. En effet, les règles de transition de l'automate d'arbre qui accepte la classe de documents (sous forme d'arbre) décrit par un schéma utilisent des expressions régulières pour associer un état à un nœud de l'arbre XML. Notre méthode considère donc que les mises à jour sont exécutées sur un groupe de nœuds attachés à un même nœud père. Ainsi, la mise à jour du schéma est faite en proposant des expressions régulières candidates pour remplacer l'expression régulière de la règle de transition de $\mathcal{A}$ qui n'a pas pu associer un état valide à l'arbre mis à jour.

Ce rapport est constitué de quatre parties. Dans la partie I, nous présentons les définitions de base nécessaires dans la suite du rapport. Le chapitre 2 introduit le langage XML et la représentation des documents XML sous la forme d'arbres d'arité non-bornée. Dans le chapitre 3, nous présentons plusieurs langages de définition de schémas XML en insistant sur les DTD, le formalisme utilisé dans la contribution. Avant de présenter ces langages et les schémas qu'ils décrivent, nous introduisons les grammaires d'arbres qui modélisent les schémas XML.

Dans la partie II, nous présentons une synthèse des différents domaines de recherches liés à nos contributions. Ainsi, nous parlons, dans le chapitre 4, des différentes méthodes de validations de documents XML, aussi bien que des validations incrémentales de documents XML, c'est-à-dire, les validations qui prennent en considération une partie d'un document et, éventuellement, des informations auxiliaires obtenues par des validations précédentes. Dans le chapitre 5, nous présentons des méthodes utilisées pour conserver la validité des documents XML soit en modifiant les documents, soit en modifiant les schémas. Ces méthodes ne sont pas forcément proposées pour faire évoluer des schémas XML, cependant elles peuvent d'une façon plus ou moins efficace être utilisées dans ce cadre. La section 5.3, dans le chapitre 5, présente aussi des primitives de mises à jour pour les schémas XML.

La partie III présente les contributions de cette thèse et elle est divisée en trois chapitres :

- Dans le chapitre 7, nous présentons notre méthode de vérification de validité des documents XML.
- Dans le chapitre 8, la méthode pour guider l'évolution des schémas XML est présentée. L'évolution de schéma est déclenchée par une mise à jour invalide qui représente la spécification de l'évolution désirée (elle est exécutée par l'utilisateur administrateur de l'application). Cette méthode s'appuie sur la méthode de validation du chapitre 7.
- Le chapitre 9 présente une extension de la méthode du chapitre 8 : on y considère une série de mises à jour.

Finalement, dans la partie IV, le chapitre **Conclusions et perspectives** résume les principales contributions et présente un certain nombre de directions de recherche pouvant être envisagées à partir de nos travaux. Les annexes regroupent les démonstrations des théorèmes ainsi que l'algorithme de construction d'un automate d'arbre à partir d'un schéma XML modélisé par une DTD.

Première partie

Préliminaires

Chapitre 2

XML

Dans ce chapitre nous introduisons les principes du langage XML pour lequel nous allons proposer un modèle de validation et une méthode pour guider l'évolution de schémas.

2.1 Le langage XML

XML est l'acronyme d'eXtensible Markup Language. Il est un sous-ensemble du langage SGML[1] (Standard Generalized Markup Language) [Gol91] et il a été approuvé par le World Wide Web Consortium (W3C)[W3C00] en février 1998. XML n'est pas réellement un langage, c'est un méta-langage à balise, utilisé pour définir des langages de description (*e.g.*, XML-Schema, XUpdate, etc.). Un document XML est composé de blocs de textes structurés par des balises de début et de fin, par exemple, <foo> foo </foo>. Ainsi, un document XML peut non seulement contenir des données, mais aussi leur structure. En utilisant XML, le document transporte donc de l'information à propos des données qu'il contient. Ces données sont dans le format texte et ainsi, les documents XML sont indépendants des plate-formes.

Le fichier XML est structuré en *éléments*. Les éléments peuvent contenir du texte et éventuellement d'autres éléments. L'ensemble des données du document XML est contenu dans un élément unique appelé *racine*, élément qui contient tous les autres éléments.

Le langage XML est extensible, c'est-à-dire, que l'on peut définir les noms de balises et la structure des balises librement dans un document XML. La seule contrainte à respecter est celle de la bonne formation. Pour qu'un document XML soit bien-formé, il doit suivre les règles suivantes :

- Tout nom de balise doit commencer par une lettre ou un souligné (*underscore*) suivi, optionnellement, par des lettres, chiffres, traits d'union, points, deux points ou soulignés. La chaîne "xml", quelle que soit sa casse, en début des noms est interdite (réservée pour de futurs usages).
- Toute balise ouverte doit être fermée, soit par $/>$ si l'élément est vide, soit par la balise fermante correspondante ($< /nomBalise >$).
- Ces deux balises (ouvrante et fermante) doivent être correctement imbriquées entre les balises de l'élément parent (pas de recouvrement).

Par son format simple, non-ambigu, indépendant de toute plate-forme et générique, XML

[1] SGML est un langage de codage de données dont l'objectif est de permettre, dans un échange entre systèmes informatiques, de transférer, en même temps, des données textuelles et leurs structures. SGML a été implanté et standardisé par l'Organisation Internationale de Normalisation (ISO) en 1986.

```
<Universite Nom="Université de Moscou">
 <Laboratoire>
   <Nom> Informatique </Nom>
   <Chercheur CId="C001">
     <Nom> Victor M. Glushkov </Nom>
     <Titre> Professeur </Titre>
   </Chercheur>
   <Publication>
     <Sujet> Automates </Sujet>
     <Annee> 1961 </Annee>
     <Revue>
        <Nom> Revue Russe de Recherche </Nom>
        <TArticle idAuts="C001">
               Théorie Abstraite des Automates
        </TArticle>
     </Revue>
   </Publication>
 </Laboratoire>
</Universite>
```

FIG. 2.1 – Un document XML qui modélise les laboratoires de recherche d'une université.

facilite les échanges de données de types très différents entre des applications hétérogènes et les traitements par des outils standardisés. On peut définir des règles pour la création de documents XML et, dans ce cas, on utilise les langages de schémas. Si un document XML X est conforme à un schéma, X est dit valide. Dans le chapitre 3 nous présentons les langages de schémas et donc, la notion de document valide.

La figure 2.1 présente un exemple d'un document XML bien-formé qui stocke les données des laboratoires de recherches attachés à une université. Dans la suite, nous présentons brièvement la syntaxe du langage XML (voir [W3C00] pour plus de détails).

Dans notre exemple, $<Universite>$, $<Laboratoire>$ et $<Chercheur>$ sont des balises; elles permettent de délimiter les éléments d'un document. Ainsi, on a les éléments $Universite$, $Laboratoire$ et $Chercheur$ représentés par les balises précédentes. Dans la figure 2.1, nous trouvons aussi des formes $<TArticle\ idAuts = \text{“}C001\text{”}>$. Dans ce cas, $idAuts$ est un attribut attaché à l'élément $TArticle$. Un attribut est une paire (nom, $valeur$) qui permet de caractériser un élément. Le rôle des attributs consiste en effet à fournir des informations supplémentaires sur un élément, sans que cet élément n'ait besoin de stocker explicitement ces informations. Un élément peut avoir plusieurs attributs, cependant les noms des attributs doivent être uniques. Les paires d'attributs (nom, $valeur$) définis pour un élément sont séparées par un espace.

Le contenu d'un élément e est appelé le modèle de contenu de e. Un élément peut avoir le modèle de contenu suivant :

1. Une valeur texte, comme l'élément Nom;
2. D'autres éléments, comme l'élément $Chercheur$ et, dans ce cas, les éléments Nom et $Titre$ qui apparaissent dans $Chercheur$ sont appelés sous-éléments de $Chercheur$; ou
3. Le deux : des valeurs texte et des sous-éléments et, dans ce cas, le modèle de contenu est appelé mixte.

La syntaxe XML convient pour décrire les données semi-structurées [ABS00, Suc98]. En fait, depuis quelques années, XML remplace les formalismes qui modélisaient les données semi-structurées, *e.g.*, OEM (Object Exchange Model) [PGW95] et UnQL (Unstructured Query Language) [BDHS96]. À la différence des modèles semi-structurés de données, XML ne provient pas de la communauté des bases de données mais de la communauté des documents. Pour cette communauté, les données XML sont des parties de textes et les balises servent à décrire le contexte, la structure et la sémantique de ces textes. En revanche, la communauté des bases de données a adopté XML comme un modèle pour les données semi-structurées et tend à transférer les résultats établis pour des données régulières (*e.g.*, base de données relationnelles) aux modèles semi-structurés représentés par XML. Ainsi, la communauté des bases de données cherche par exemple à trouver des solutions pour le stockage de très grosses bases de données XML, l'intégration de données de sources hétérogènes en utilisant XML, l'exportation de vues XML à partir de bases de données relationnelles, la transformation des données pour l'échange de données entre applications, etc. (voir entre autres [FSW99]). Dans notre travail, nous nous intéressons à l'utilisation de XML comme un modèle de données semi-structurées dans le contexte des bases de données. Plus particulièrement, dans le cadre de la vérification de la validité de documents XML et des mises à jour de leurs schémas (induite par des mises à jour des documents).

Dans la prochaine section nous définissons la représentation des documents XML la plus utilisée, à savoir les arbres XML.

2.2 Représentation des documents XML sous forme d'arbre

Le langage complet XML a beaucoup de caractéristiques, cependant l'abstraction la plus simple pour un document XML est un arbre étiqueté (étiquettes dans les nœuds) et, éventuellement, avec des valeurs associées aux nœuds feuilles [Via03]. L'ensemble des nœuds de l'arbre d'un document est muni d'un ordre : l'ordre du document qui est l'ordre de lecture, dans le document XML, des constituants représentés par chaque nœud. Par exemple, la figure 2.2 présente le document XML de la figure 2.1 sous la forme d'un arbre étiqueté t. Comme le montre la figure 2.2, l'élément le plus externe d'un document XML est la racine de l'arbre (*i.e.*, l'élément *Universite*) et chaque élément, à partir de la racine, a ses attributs et sous-éléments comme fils. Remarquer que les nœuds qui représentent les attributs sont préfixés avec @.

La majorité des travaux sur XML qui utilisent la représentation par un arbre ne modélisent pas les attributs comme nœuds indépendants dans l'arbre [Chi00, PV00, TIHW01]. Dans notre approche les attributs d'un élément e d'un document XML sont représentés aussi par des nœuds dans l'arbre XML, fils du nœud qui représente e. Par exemple, l'attribut *idAuts* de l'élément *TArticle* apparaît dans t comme un nœud fils du nœud étiqueté *TArticle*. Les nœuds qui représentent des valeurs sont représentés par des nœuds étiquetés *data*.

Les arbres XML sont des arbres d'arité non-bornée, c'est-à-dire, on ne connaît pas le nombre de fils d'un nœud *a priori*. Dans la figure 2.2, par exemple, les nœuds étiquetés *Laboratoire* peuvent avoir différents nombres de fils étiquetés *Chercheur*. Remarquons que le nombre de fils n'est pas connu *a priori* mais il est fini.

Dans la suite, nous présentons la notion d'arbre XML d'une manière plus formelle.

2.2.1 Arbre

Les arbres d'arité bornée sont étudiés depuis longtemps [CDG+97, GS97, Tho97] alors que l'intérêt pour les arbres d'arité non-bornée est plutôt récent [AMN+01, BW98b, MLM01, Nev02a, PV00], motivé par XML.

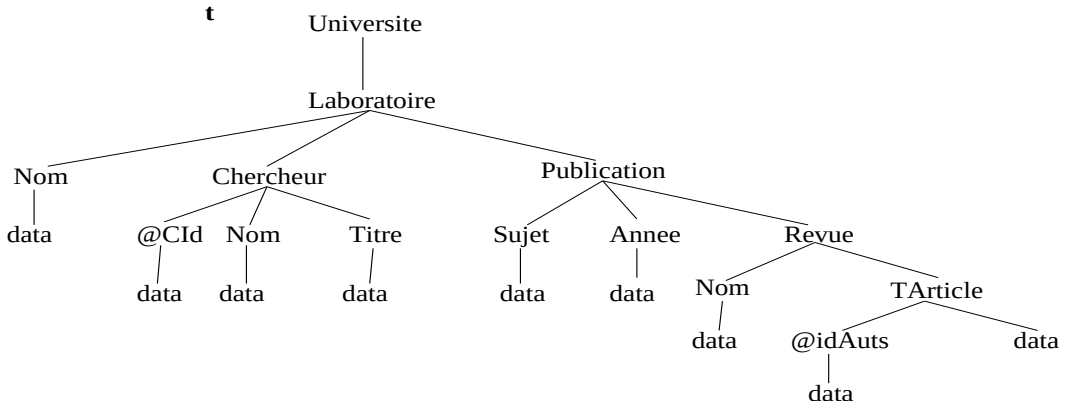


FIG. 2.2 – L'arbre XML qui représente le document XML de la figure 2.1.

Pour définir notre notion d'arbre, nous considérons un ensemble fini de symboles (étiquettes), appelé alphabet et noté Σ. Nous notons U l'ensemble $\mathbb{N}^*$ de toutes les séquences d'entiers positifs ou nuls. L'ensemble U muni du produit de concaténation est un monoïde et son élément neutre est noté ε.

Définition 2.2.1 - La relation de préfixe : La *relation de préfixe* dans U, notée $\preceq$, est définie comme suit : $u \preceq v$ si $uw = v$ pour un $w \in U$. Étant donné un sous-ensemble fini $D \subseteq U$, nous disons que D est *fermé pour l'ordre* $\preceq$ si pour tous u et v de U tels que $u \preceq v$, $v \in D$ implique $u \in D$. □

Les exemples présentés dans ce travail utiliserons toujours des entiers inférieurs ou égaux à 9 pour représenter les positions dans les arbres XML. Alors, nous allons écrire la position d'un nœud quelconque d'un arbre sans séparateur entre les entiers de chaque niveau.

Définition 2.2.2 - L'arbre t étiqueté par Σ: Un arbre t étiqueté par Σ (ou simplement un arbre) est une application

$$t : dom(t) \rightarrow \Sigma$$

où $dom(t) \subseteq U$ est un ensemble non vide et fermé pour $\preceq$ qui satisfait :

$$(\forall j \geq 0)(uj \in dom(t)) \Rightarrow (\forall i)(0 \leq i \leq j \Rightarrow ui \in dom(t)).$$

L'ensemble $dom(t)$ est aussi appelé l'ensemble des positions de t. La position de la racine de t est représentée par ε. Nous écrivons $t(v) = a$, pour chaque $v \in dom(t)$, pour indiquer que le symbole $a \in \Sigma$ est associé au nœud dans la position v. Un *arbre vide* t est tel que $dom(t) = \emptyset$. □

La figure 2.3 présente l'arbre XML de la figure 2.2 avec ses positions. Remarquons que la position du noeud racine est ε et que l'ensemble des positions de l'arbre est fermé par l'ordre $\preceq$, puisque la position d'un noeud est toujours préfixée par celle de son père. Il en est de même pour l'ensemble des positions intérieures.

Nous présentons aussi la notion de frontière d'un arbre fini qui est l'ensemble des positions des feuilles.

Définition 2.2.3 - La frontière d'un arbre fini t: Étant donné un arbre t, la *frontière* de t, notée par $fr(t)$ est définie comme suit :

$$fr(t) = \{u \in dom(t) \mid (\forall i \in \mathbb{N})\ ui \notin dom(t)\}.$$

Si t est vide alors $fr(t) = \emptyset$. □

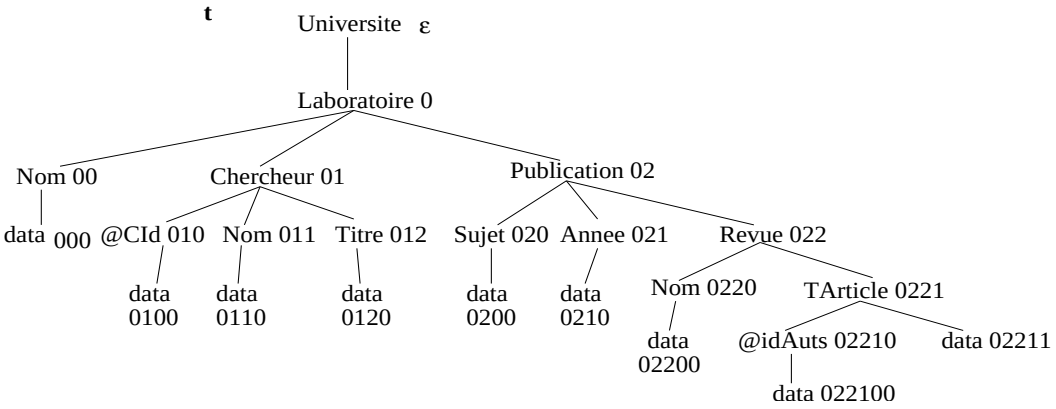


FIG. 2.3 – L'arbre XML de la figure 2.2 avec ses positions.

Exemple 2.2.1 Étant donné l'arbre XML de la figure 2.3, nous avons $\Sigma = \{Universite, Laboratoire, Nom, Chercheur, Publication, CId, Titre, Sujet, Annee, Revue, TArticle, IdAuts, data\}$. Nous écrivons $t(\varepsilon) = Universite$ pour indiquer que le symbole dans Σ associé à la position ε de t est $Universite$. De même, nous pouvons écrire, par exemple, $t(02) = Publication$ et $t(0221) = TArticle$. Notons que les éléments et les attributs qui ont des valeurs associées aux textes ont des fils étiquetés *data*. Par exemple, $t(012) = Titre$ et $t(0120) = data$. La frontière de t est l'ensemble des positions des feuilles de t. □

Dans le cadre des mises à jour des arbres XML, la définition de la frontière d'insertion est importante car c'est l'ensemble des positions qui n'appartiennent pas à $dom(t)$ mais qui peuvent être utilisées pour insérer de nouveaux sous-arbres dans t.

Définition 2.2.4 - La frontière d'insertion d'un arbre fini t: Étant donné un arbre t, la *frontière d'insertion* de t, notée par $fr^{ins}(t)$ est définie comme suit :

$$fr^{ins}(t) = \{ui \notin dom(t) \mid i \in \mathbb{N} \wedge u \in dom(t) \wedge [(i = 0) \vee ((i \neq 0) \wedge u(i-1) \in dom(t)]\}.$$

Si t est vide alors $fr^{ins}(t) = \{\varepsilon\}$. □

Exemple 2.2.2 Étant donné l'arbre XML de la figure 2.3, la frontière d'insertion de t est ici $fr^{ins}(t) = \{1, 03, 001, 013, 023, 0000, 0101, 0111, 0121, 0201,0211,0222,01000,01100, 01200, 02000, 02100, 02201, 02212, 022000, 0221000, 022110\}$. La position 0222 par exemple appartient à la frontière d'insertion parce que le noeud étiqueté $TArticle$ de position 0221 n'a pas de frère droit; la position 0222 est donc disponible pour y insérer un nouveau noeud. □

Les documents XML sont des arbres d'arité non-bornée ordonnés et étiquetés par un alphabet fini Σ. L'alphabet d'un arbre XML peut être déterminé par un schéma qui sera l'objet du prochain chapitre.

Chapitre 3

Schémas

Dans notre travail, nous nous intéressons seulement aux documents XML qui satisfont les contraintes des schémas. Dans ce chapitre nous introduisons les schémas XML, principalement les DTD (*Document Type Definition*) utilisées dans notre approche. Les notations et définitions présentées sont inspirées des définitions et notations proposées dans [CDG+97, FL01, HMU01, Nev99, Woo87, WF74]. Nous présentons aussi les grammaires d'arbres qui sont utilisées pour introduire les schémas XML.

3.1 Notions préliminaires

3.1.1 Ensembles, mots et langages

Si C est un ensemble fini, alors $|C|$ dénote le cardinal de C. Un alphabet est un ensemble fini de symboles. Un *mot* w (ou une chaîne de caractères) sur un alphabet Σ est composé de 0 ou plusieurs symboles de Σ. Le symbole ε dénote le mot vide qui est composé de 0 symbole. L'ensemble infini de tous les mots finis sur l'alphabet Σ est noté par Σ^*. Dans cette thèse nous utilisons le terme mot pour désigner une chaîne de caractères ou une séquence de symboles appartenant à un alphabet fixé Σ.

Soit w un mot (ou chaîne de caractères). Nous posons que la première position de w est la position 0 et que $|w|$ dénote la taille (nombre de positions) de w. De plus nous utilisons la notation suivante [WF74] :

1. $w[i]$ est le i-ème caractère (symbole) de w,
2. $w[i:j]$ représente les caractères de la i-ème jusqu'à la j-ème position (inclus) de w (*i.e.*, $w[i:j] = w[i]w[i+1]\dots w[j]$), et
3. $w[i:j] = \varepsilon$, si $i > j$.

Si u et v sont des mots sur un alphabet Σ, alors leur concaténation uv (ou $u \cdot v$) est encore un mot sur Σ. Un mot u est un préfixe du mot v s'il existe un mot w tel que $uw = v$. De la même manière, un mot u est un suffixe du mot v s'il existe un mot w tel que $wu = v$.

Un langage est un sous-ensemble de Σ^*. Les opérations suivantes sont définies sur les langages :

- La concaténation de deux langages L_1 et L_2 est définie comme le langage $L_1L_2 = \{uv \mid u \in L_1 \wedge v \in L_2\}$.
- L'intersection de deux langages L_1 et L_2 est définie comme le langage $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$.

- L'union de deux langages L_1 et L_2 est définie comme le langage $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$.
- La fermeture L^* d'un langage L est définie comme le langage $L^* = \sum_{i=0}^{\infty} L^i$, où L^0 contient seulement le mot vide ε et L^i les mots de taille $i > 0$, *i.e.*, pour tous les mots $w \in L^i$, $|w| = i$.
- La fermeture positive L^+ d'un langage L est définie comme le langage $L^+ = \sum_{i=1}^{\infty} L^i$.

Étant donné un langage L, pour connaître les mots qu'il contient, il faut décrire L. Les grammaires formelles sont utilisées pour décrire les langages. La notion de grammaire formelle a été introduite par Chomsky [Cho56]. Une grammaire G est un quadruplet (N, Σ, P, I) où :

- Σ est l'alphabet des symboles terminaux (symboles apparaissant dans les mots générés);
- N est l'alphabet des symboles non-terminaux (symboles n'apparaissant pas dans les mots générés);
- $I \in N$ est le symbole initial;
- P est un ensemble fini de règles de production de la forme $\alpha \rightarrow \beta$, où $\alpha \in N_\Sigma^+$ et $\beta \in N_\Sigma^*$ avec $N_\Sigma = (N \cup \Sigma)$.

La signification intuitive de ces règles de production est que la suite non vide de symboles terminaux ou non-terminaux $\alpha \in N_\Sigma^+$ peut être remplacée par la suite, éventuellement vide, de symboles terminaux ou non-terminaux $\beta \in N_\Sigma^*$.

Une grammaire génère les mots d'un langage en réécrivant le symbole initial I. Si $\alpha \rightarrow \beta$ est une règle de production de la grammaire G, on peut réécrire (ou dériver) le mot "a α b" en "a β b" en appliquant $\alpha \rightarrow \beta$ sur "a α b".

Chomsky a défini une classification des grammaires formelles en quatre types dépendants de la forme de leurs règles de production. Soient $\alpha, \beta \in N_\Sigma^*$, $A \in N$, $v \in N_\Sigma^+$ et $a, b \in \Sigma$. Le tableau 3.1 présente cette classification.

Tab. 3.1 – Classification de Chomsky

Type	**Forme de la règle de production**	**Remarques**
type 0	$\alpha \rightarrow \beta$	aucune restriction sur les règles de production
type 1	$\alpha A \rightarrow \alpha v \beta$	dépendante du contexte
type 2	$A \rightarrow \alpha$	indépendante du contexte (hors contexte)
type 3	$A \rightarrow aB$ ou $A \rightarrow Ba$	régulière (équivalente aux expressions régulières)

Les langages réguliers (type 3) sont équivalents aux expressions régulières. Les expressions régulières sont très utilisées dans cette thèse, aussi, nous les décrivons dans la prochaine section.

3.1.2 Langages réguliers, expressions régulières, automates d'états finis

Un langage régulier ou langage rationnel ou encore langage de type 3 dans la hiérarchie de Chomsky [Cho56], est un langage formel que l'on peut définir grâce à une expression régulière sur un alphabet.

Définition 3.1.1 - Expression régulière [HMU01] : Une *expression régulière* E sur un alphabet Σ et son langage associé $L(E)$ sont définis de façon inductive comme suit:

1. Les constantes ε et $\emptyset$ sont des expressions régulières et elles dénotent les langages $\{\varepsilon\}$ et $\emptyset$, respectivement, c'est-à-dire, $L(\varepsilon) = \{\varepsilon\}$ et $L(\emptyset) = \emptyset$.
2. Si $a \in \Sigma$, alors a est une expression régulière qui dénote le langage $L(a) = \{a\}$.

3. Si E et F sont des expressions régulières, alors $E + F$ est une expression régulière (notée aussi par $E|F$) qui dénote l'union de $L(E)$ et $L(F)$, *i.e.*, $L(E + F) = L(E) + L(F)$.
4. Si E et F sont des expressions régulières, alors EF est une expression régulière qui dénote la concaténation de $L(E)$ et $L(F)$, *i.e.*, $L(EF) = L(E)L(F)$. L'opérateur $\cdot$ peut aussi être utilisé pour dénoter la concaténation.
5. Si E est une expression régulière, alors E^* est une expression régulière qui dénote la fermeture de $L(E)$ (étoile de Kleene), *i.e.*, $L(E^*) = (L(E))^*$.
6. Si E est une expression régulière, alors E^+ est une expression régulière qui dénote la fermeture positive de $L(E)$, *i.e.*, $L(E^+) = L(E)L(E^*)$.
7. Si E est une expression régulière, alors $E?$ est une expression régulière qui dénote le langage $L(E)$ qui de plus, accepte le mot vide, *i.e.*, $L(E?) = L(E) \cup \varepsilon$.
8. Si E est une expression régulière, alors (E), l'expression régulière avec parenthèses, est aussi une expression régulière qui dénote le même langage que E. □

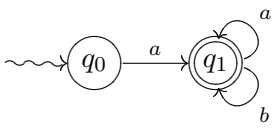
Pour éviter le recours excessif aux parenthèses, on pose que l'étoile de Kleene (et la fermeture positive $^+$) a la plus haute priorité, suivie par la concaténation puis par l'union. Par exemple, $(a\ b)c$ peut s'écrire $a\ b\ c$ et $a|(b(c^*))$ peut s'écrire $a|bc^*$. Si on veut changer la priorité des opérateurs, alors on utilise les parenthèses.

Un langage L est régulier s'il existe une expression régulière E telle que $L = L(E)$. Un mot w est une instance d'une expression régulière si w appartient au langage défini par E. Le problème de savoir comment déterminer si un mot w est dans un langage régulier L a été traité par Kleene, dans [Kle65] : il a montré que les automates d'états finis et les expressions régulières définissent la même classe de langages.

Définition 3.1.2 Automates d'états finis déterministes M: Un automate d'états finis déterministe M est un quintuplet $M = (Q, \Gamma, \Delta, q_0, F)$, où (i) Q est un ensemble non vide d'états, (ii) Γ est l'ensemble des symboles d'entrée, (iii) $\Delta : Q \times \Gamma \to Q$ est la fonction de transition, (iv) $q_0 \in Q$ est l'état initial de M et (v) $F \subseteq Q$ est l'ensemble des états finaux. □

La fonction de transition Δ est une application qui fait correspondre un couple de $Q \times \Gamma$ à un état de Q. Elle a la forme $\delta(q, a) = q'$, où $q, q' \in Q$ et $a \in \Gamma$, et signifie que depuis q, en utilisant le symbole a, on atteint l'état q' dans M. La fonction de transition peut être étendue[1] à $Q \times \Gamma^* \to Q$, où $\hat{\delta}(q, \varepsilon) = q$ et $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$. Nous allons noter $\perp$ pour représenter un état non connu par l'automate (*i.e.*, un état invalide).

Exemple 3.1.1 Considérons le langage régulier L sur l'alphabet $\Gamma = \{a, b\}$ formé des mots qui commencent par le symbole a et qui ont un nombre quelconque de a et b dans la suite. Ce langage peut être dénoté par l'expression régulière $E = a(a|b)^*$. Il est reconnu par l'automate d'états finis M sur $\Gamma = \{a, b\}$, avec $Q = \{q_0, q_1\}$, $\Gamma = \{q_0\}$, $F = \{q_1\}$ et δ définie par : $\delta(q_0, a) = q_1$, $\delta(q_1, a) = q_1$, $\delta(q_1, b) = q_1$.
Ci-dessous nous montrons une représentation graphique de l'automate M où l'état final est noté par un double cercle et l'état initial par un cercle avec une flèche entrante.



□

Dans la prochaine section, nous allons présenter les grammaires d'arbres qui modélisent les langages de schémas XML (sujet important dans le cadre de cette thèse).

[1] La notation δ^* est aussi utilisée pour représenter les transitions étendues.

3.1.3 Langage d'arbres et grammaire d'arbres

Comme les documents XML peuvent être vus comme des arbres, on peut considérer que les grammaires d'arbres sont les mécanismes les plus naturels pour décrire ces documents. Dans les langages formels de mots (comme le montrent les sections 3.1.1 et 3.1.2), les automates d'états finis et les grammaires formelles de mots sont utilisés pour reconnaître et pour définir les langages, respectivement. Dans le cas des arbres, les automates d'arbres et les grammaires d'arbres sont utilisés. Les grammaires régulières d'arbres peuvent être considérées comme des grammaires du type 2 dans la hiérarchie de Chomsky [CDG$^+$97, GS97]. Les schémas XML sont des grammaires d'arbres [MLMK04, Via01]. Dans cette section nous présentons la classification des grammaires d'arbres permettant de décrire des arbres valides et de classer les langages de schémas par rapport à leur pouvoir d'expression. Ces définitions sont celles de [MLMK04] et elles portent sur les arbres d'arité non-bornée.

Grammaire d'arbres régulière

Une grammaire d'arbres régulière est un mécanisme pour générer des arbres.

Définition 3.1.3 Grammaire d'arbres régulière (RTG[2]) et langage [MLMK04] : Une *grammaire d'arbres régulière* est un quadruplet $G = (N, \Sigma, P, I)$, où :
- N est un ensemble de symboles non-terminaux (appelés aussi types);
- Σ est un ensemble de symboles terminaux;
- P est un ensemble de règles de productions de la forme $X \rightarrow \mathbf{a}\ (r)$, où $X \in N$, $\mathbf{a} \in \Sigma$ et r est une expression régulière sur N (r est le modèle de contenu de $\mathbf{a}$);
- I est un ensemble de symboles initiaux ($I \subseteq N$).

Un *langage d'arbres régulier* est l'ensemble des arbres dérivés d'une grammaire d'arbres régulière. □

Comme nous utilisons les grammaires d'arbres dans le contexte XML et nous représentons les données dans les arbres XML, nous allons utiliser le symbole terminal spécial **data** pour représenter les données. De plus, on note les symboles terminaux en gras, et les symboles non-terminaux (les types) et les expressions régulières en italique. La séquence vide de symboles non-terminaux est notée par ε.

Exemple 3.1.2 La grammaire $G_1 = (N, \Sigma, P, I)$ où :
- $N = \{Publication, Article1, Article2, Conference, Auteur, data\}$
- $\Sigma = \{$**Publication**, **Article**, **Conference**, **Auteur**, **data**$\}$
- $P = \{Publication \rightarrow$ **Publication** $(Article1\ Article2^*)$, $Article1 \rightarrow$ **Article** $(Auteur\ Conference)$, $Article2 \rightarrow$ **Article** $(Auteur)$, $Conference \rightarrow$ **Conference** $(data)$, $Auteur \rightarrow$ **Auteur** $(data)$, $data \rightarrow$ **data** $(\varepsilon)\}$.
- $I = \{Publication\}$

décrit l'arbre XML t de la figure 3.1, *i.e.*, $t \in L(G_1)$. Les symboles terminaux correspondent aux noms des éléments et des attributs du document. Le symbole terminal *data* correspond aux données du document et, dans ce cas, il peut représenter le nom de l'auteur de l'article et le nom de la conférence où l'article a été soumis. Remarquer que le membre gauche, le membre droit et le modèle de contenu de la première règle de production sont *Publication*, **Publication** $(Article1\ Article2^*)$ et $(Article1\ Article2^*)$, respectivement. Remarquer aussi que les nœuds étiquetés *Article* dans t peuvent être construits par la règle de production de $Article1$ ou de $Article2$. Pour savoir laquelle a été utilisé il faut analyser les fils des nœuds étiquetés *Article*. Le nœud n étiqueté *Article* plus à gauche a été généré par la règle de production de $Article1$ car

[2] Regular Tree Grammar

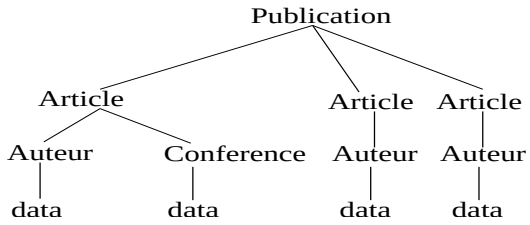


FIG. 3.1 – L'arbre généré par la grammaire régulière d'arbres de l'exemple 3.1.2.

les fils de n sont décrits par le modèle de contenu de la règle de $Article1$, c'est-à-dire, $Auteur\ Conference$. □

Grammaire d'arbres régulière locale

Les grammaires d'arbres régulières locales sont une spécialisation des grammaires d'arbres régulières. En effet, les grammaires d'arbres régulières locales sont des grammaires qui ont une contrainte pour générer leurs arbres. Cette contrainte est fondée sur la définition de la concurrence entre symboles non-terminaux énoncée ci-dessous.

Définition 3.1.4 Concurrence entre symboles non-terminaux [MLMK04] : Soit une grammaire d'arbres (régulière) $G = (N, \Sigma, P, I)$. Deux non-terminaux A et B dans N sont concurrents s'il existe deux règles de production $A \rightarrow \mathbf{c}\ (r)$ et $B \rightarrow \mathbf{c}\ (r')$ dans P telles que $A \neq B$ et $r \neq r'$. Dans ce cas les non-terminaux A et B sont en concurrence pour le symbole $\mathbf{c}$. □

Définition 3.1.5 Grammaire d'arbres régulière locale (LTG[3]) et langage [MLMK04] : Une *grammaire d'arbres régulière locale* est une grammaire d'arbres régulière sans non-terminaux concurrents. Un *langage d'arbres régulier local* est l'ensemble des arbres dérivés d'une grammaire d'arbres régulière locale. □

Remarquer que les LTG sont également des grammaires d'arbres régulières (l'ensemble des grammaires d'arbres régulières locales est inclus dans l'ensemble des grammaires d'arbres régulières).

Exemple 3.1.3 Dans l'exemple 3.1.2 les règles $Article1 \rightarrow$ **Article** $(Auteur\ Conference)$, $Article2 \rightarrow$ **Article** $(Auteur)$ de la grammaire G_1 sont en concurrence pour le symbole **Article**. Pour transformer G_1 en une grammaire d'arbres régulière locale il faut enlever la concurrence dans G_1. La grammaire $G_2 = (N, \Sigma, P, I)$ est une grammaire locale :

- $N = \{Publication, ArticlePublie, Article, Conference, Auteur, data\}$
- $\Sigma =$ {**Publication**, **ArticlePublie**, **Article**, **Conference**, **Auteur**, **data**}
- $P = \{Publication \rightarrow$**Publication** $(ArticlePublie\ Article^*)$, $ArticlePublie \rightarrow$ **ArticlePublie** $(Auteur\ Conference)$, $Article \rightarrow$ **Article** $(Auteur)$, $Conference \rightarrow$ **Conference** $(data)$, $Auteur \rightarrow$ **Auteur** $(data)$, $data \rightarrow$ **data** $(\varepsilon)\}$.
- $I = \{Publication\}$

Remarquer que les règles de production des symboles non-terminaux $Article1$ et $Article2$ ont été remplacées par les règles $ArticlePublie \rightarrow$ **ArticlePublie** $(Auteur\ Conference)$, $Article \rightarrow$ **Article** $(Auteur)$, respectivement. □

[3]Local regular Tree Langage

Grammaire d'arbres régulière à type unique

Les grammaires d'arbres régulières à type unique sont aussi une spécialisation des grammaires d'arbres régulières, cependant elles sont moins restreintes que les grammaires d'arbres régulières locales. Elles acceptent que deux symboles non-terminaux soient en concurrence pour un même symbole mais ces deux symboles ne peuvent pas apparaître dans le même modèle de contenu d'une troisième règle de production. La restriction imposée aux grammaires d'arbres régulières à type unique est présentée dans la définition ci-dessous.

Définition 3.1.6 Grammaire d'arbres régulière à type unique (STG[4]) et langage [MLMK04] : Une *grammaire d'arbres régulière à type unique* est une grammaire d'arbres régulière telle que
(i) pour chaque règle de production $X \rightarrow \mathbf{a}\ (r)$, les non-terminaux dans l'expression régulière r ne sont pas concurrents et
(ii) les symboles dans I ne sont pas concurrents.
Le langage d'arbres L est un langage d'arbre à type unique si les arbres $t \in L$ sont dérivés d'une grammaire d'arbres régulière à type unique. □

Exemple 3.1.4 Reprenons l'exemple 3.1.2 : la grammaire G_1 de l'exemple a les types $Article1$ et $Article2$ qui sont concurrents (du au symbole **Article**) et ils apparaissent dans la même expression régulière de la règle $Publication \rightarrow$ **Publication** $(Article1\ Article2^*)$. Alors G_1 n'est pas une grammaire d'arbres régulière à type unique. Étant donné la grammaire $G_3 = (N, \Sigma, P, I)$ où :

- $N = \{Publication, Articles, Brevet, Revue, AuteurR, AuteurB, Nom, Titre, Annee, Emetteur, data\}$
- $\Sigma =$ {**Publication**, **Articles**, **Brevet**, **Revue**, **Auteur**, **Nom**, **Titre**, **Annee**, **Emetteur**, **data**}
- $P = \{Publication \rightarrow$**Publication** $(Articles^*\ Brevet^*)$, $Articles \rightarrow$ **Articles** $(Revue)$, $Brevet \rightarrow$ **Brevet** $(AuteurB)$, $Revue \rightarrow$ **Revue** $(AuteurR)$, $AuteurR \rightarrow$ **Auteur** $(Nom\ Titre\ Annee)$, $AuteurB \rightarrow$ **Auteur** $(Nom\ Titre\ Emetteur)$, $Nom \rightarrow$ **Nom** $(data)$, $Titre \rightarrow$ **Titre** $(data)$, $Annee \rightarrow$ **Annee** $(data)$, $Emetteur \rightarrow$ **Emetteur** $(data)$, $data \rightarrow$ **data** $(\varepsilon)\}$.
- $I = \{Publication\}$.

G_3 est une grammaire régulière d'arbre à type unique même si les types $AuteurR$ et $AuteurB$ sont concurrents (du au symbole **Auteur**). Ils n'apparaissent pas dans la même expression régulière d'une règle de production de G_3. □

Les grammaires d'arbres permet de classer les formalismes de schémas XML selon leur pouvoir d'expression. L'expressivité des grammaires d'arbre peut être classée comme suit [MLMK04] :

$$\text{LTG} \subset \text{STG} \subset \text{RTG}.$$

Si on fait une correspondance entre les grammaires d'arbres ci-dessus et les arbres qu'elles dérivent, alors on peut les décrire comme suit :

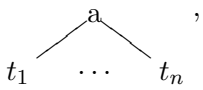
- Dans les arbres dérivés d'une LTG, tous les nœuds étiquetés a sont toujours associés à la même règle de production.
- Dans les arbres dérivés d'une STG, les nœuds étiquetés a peuvent être associés à des différentes règles de production, cependant leurs pères ne peuvent pas avoir la même étiquette.
- Dans les arbres dérivés d'une RTG, il n'existe aucune restriction pour les règles de production par rapport à l'alphabet de types (hormis le modèle de contenu qui doit être modélisé par une expression régulière).

[4] Single-type Tree Grammar

Ainsi, les schémas XML qui produisent des arbres XML valides sont des grammaires d'arbres et ils peuvent être classés par rapport au pouvoir d'expression des grammaires d'arbres correspondantes. Avant d'introduire les langages de schémas, nous présentons l'un des mécanismes utilisés pour reconnaître les langages d'arbres, les automates d'arbre. Dans cette thèse, nous utilisons les automates d'arbres ascendants d'arité non-bornée pour vérifier si les arbres XML appartiennent à un langage donné.

3.2 Automates d'arbres d'arité non-bornée

Les automates d'arbres décrivent les langages réguliers d'arbre [BW98b]. Comme ils ont été d'abord étudiés pour les arbres d'arité bornée (surtout les arbres binaires) [CDG+97, GS97], plusieurs auteurs voient les arbres XML comme des arbres binaires pour travailler avec les automates d'arbres d'arité bornée [FGK03, MSV00, PV03]. Cependant, les arbres XML sont naturellement des arbres d'arité non-bornée et les automates d'arbres d'arité non-bornée ont fait l'objet plus récemment de nombreux travaux [BW98b, LMM00].

Définition 3.2.1 Automate d'arbre ascendant non déterministe d'arité non bornée (NFTA)[5] [Nev99] : Un *automate d'arbre ascendant non déterministe d'arité non bornée* est un quadruplet $B = (Q, \Sigma, F, \Delta)$, où
(i) Q est un ensemble fini d'états,
(ii) Σ est l'alphabet qui étiquette les nœuds de l'arbre,
(iii) $F \subseteq Q$ est l'ensemble d'états finaux et
(iv) Δ est l'ensemble de règles de transition de la forme $\Sigma \times ExpRat(Q) \rightarrow Q$, $ExpRat(Q)$ dénote la famille des expressions régulières sur l'alphabet Q, telles que pour chaque $\delta(a, E_Q) \in \Delta$, E_Q est une expression régulière sur Q et $a \in \Sigma$.
La sémantique de B sur un arbre t, notée $\delta^*(t)$, est définie comme suit : (i) si t a juste un nœud étiqueté a alors $\delta^*(t) = \{q \mid \varepsilon \in L(E_Q), \delta(a, E_Q) \rightarrow q \in \Delta\}$, et (ii) si t a la forme 

, alors $\delta^*(t) = \{q \mid \exists q_1 \in \delta^*(t_1), \ldots, \exists q_n \in \delta^*(t_n) \wedge q_1 \ldots q_n \in L(E_Q), \delta(a, E_Q) \rightarrow q \in \Delta\}$. L'arbre t sur Σ est accepté par B si $\delta^*(t) \cap F \neq \emptyset$. Le langage d'arbre défini par B, noté $L(B)$, est composé des arbres acceptés par B. Un langage d'arbre $\mathcal{T}$ est reconnaissable s'il existe un NFTA B tel que $\mathcal{T} = L(B)$. De plus, un automate B est déterministe si pour toutes les règles $\delta(a, E_Q) \rightarrow q$ et $\delta(a, E'_Q) \rightarrow q'$ telles que $a \in \Sigma$, $q, q' \in Q$, $q \neq q'$ et $L(E_Q) \cap L(E'_Q) = \emptyset$. □

Exemple 3.2.1 Étant donné la grammaire régulière d'arbre de l'exemple 3.1.2, l'automate d'arbre d'arité non-bornée ascendant équivalent est $B = (Q, \Sigma, F, \Delta)$, où :
- $Q = \{q_{Publication}, q_{Article1}, q_{Article2}, q_{Conference}, q_{Auteur}, q_{data}\}$
- $\Sigma = \{Publication, Article, Conference, Auteur, data\}$
- $F = \{q_{Publication}\}$
- $\Delta = \{\delta(Publication, q_{Article1}\ {q_{Article2}}^*) \rightarrow q_{Publication}$,
  $\delta(Article, q_{Auteur}\ q_{Conference}) \rightarrow q_{Article1}$, $\delta(Article, q_{Auteur}) \rightarrow q_{Article2}$,
  $\delta(Conference, q_{data}) \rightarrow q_{Conference}$,
  $\delta(Auteur, q_{data}) \rightarrow q_{Auteur}$, $\delta(data, \varepsilon) \rightarrow q_{data}\}$ □

Un automate d'arbre peut être utilisé pour reconnaître tous les types de grammaires d'arbres présentés précédemment.

[5] **N**ondeterministic bottom-up **F**inite **T**ree **A**utomaton.

Dans cette thèse nous nous intéressons aux documents XML qui sont conformes (valides) par rapport à un schéma défini par une DTD. Nous considérons un document XML comme un arbre d'arité non-bornée étiqueté sur un alphabet (défini par un schéma).

3.3 Schémas

Un schéma décrit la structure des données des documents XML et sert à la validation des documents lors de l'analyse syntaxique. Un schéma peut être vu comme une grammaire que l'on utilise pour créer des documents XML bien structurés qui peuvent être traités par des applications de façon plus efficace car la structure des documents est connue *a priori*. Étant donné un schéma D et un document XML X, nous disons que X est *valide* par rapport à D, si X respecte les contraintes établies par D.

Bien que l'on trouve plusieurs langages de schémas (DTD, DSD, RELAX, Schematron, SOX [LC00]) les deux langages de schémas les plus utilisés actuellement sont les DTD (Document Type Definition) et XML-Schema [BNB04, Cho02]. Dans cette thèse, nous utilisons comme formalisme de schéma les DTD. Dans ce qui suit, nous décrivons le formalisme DTD, et ensuite, nous présentons brièvement deux autres langages : XML-Schema et RELAX-NG. Ainsi, nous allons montrer trois langages de schémas modélisés par les trois classe de grammaires d'arbres présentées dans la section 3.1.3.

3.3.1 *Document Type Definitions*

La DTD est un formalisme proposé pour SGML et permet de définir de façon formelle la structure d'un document, indépendamment de son contenu. Une DTD permet de définir une organisation des éléments entre eux et un typage de ces éléments via la notion d'attribut. Plus précisément, une DTD définit un modèle d'organisation hiérarchique des documents XML. Une DTD est équivalente à une grammaire d'arbre régulière locale [MLMK04]. La définition ci-dessous décrit la DTD telle que considérée dans cette thèse.

Définition 3.3.1 - DTD: Une *DTD* est un fichier texte ayant la forme suivante :

$$<!DOCTYPE\ firstEle\ [\textit{ensemble de déclarations d'éléments et d'attributs}]>$$

Où:

- *firstEle* est le nom de l'élément le plus externe qui peut apparaître dans un document XML. Il correspond à la racine des documents qui respectent une DTD.
- *ensemble de déclarations d'éléments et d'attributs* est la définition de la DTD. Cette définition peut avoir des *déclarations d'éléments* et des *déclarations d'attributs*:
  - La *déclaration d'éléments* a la forme suivante :

    $$<!ELEMENT\ NomÉlément\ ModèleContenu>$$

    où *NomÉlément* est le nom de l'élément à être défini et *ModèleContenu* peut être
    (i) une expression régulière *RegExp*,
    (ii) EMPTY (l'élément doit être vide),
    (iii) #PCDATA (le contenu de l'élément sera une chaîne de caractères), ou
    (iv) MIXTE, *i.e.*, le mélange des items (i) et (iii).
  - La *déclaration d'attributs* a la forme suivante :

    $$<!ATTLIST\ NomÉlément\ DéfinitionAttributs>$$

    où *NomÉlément* est le nom de l'élément père des attributs et *DéfinitionAttributs* a la forme suivante :
    - *NomAttribut Type Présence*, où :
      - *NomAttribut* est le nom de l'attribut.

- *Type* est le type de l'attribut et les types peuvent être (*i*) CDATA (une chaîne de caractères), (*ii*) ID (une chaîne de caractères sans espaces qui doit être unique par rapport aux autres chaînes du type ID - représente un identifiant unique), (*iii*) IDREF (référence à un identifiant unique du type ID précédemment créé) et (*iv*) IDREFS (liste de références à des identifiants uniques du type ID précédemment créé).
- La valeur *Présence* peut prendre les valeurs (*i*) #REQUIRED pour spécifier que l'attribut est obligatoire ou (*ii*) #IMPLIED pour spécifier que l'attribut est facultatif. □

La figure 3.2 présente un exemple de DTD à laquelle le document de la figure 2.1 est conforme. Cette DTD modélise les laboratoires qui appartiennent à une université, les chercheurs attachés et leur liste de publications.

```
<!DOCTYPE Universite [
 <!ELEMENT Universite (Laboratoire*)>
 <!ELEMENT Laboratoire (Nom,Chercheur*)>
 <!ELEMENT Nom (#PCDATA)>
 <!ELEMENT Chercheur (Nom,Titre)>
 <!ATTLIST Chercheur CId  ID #REQUIRED>
 <!ELEMENT Publication (Sujet,(Annee,Revue+)*)>
 <!ELEMENT Sujet (#PCDATA)>
 <!ELEMENT Annee (#PCDATA)>
 <!ELEMENT Revue  (Nom,TArticle)>
 <!ELEMENT Titre (#PCDATA)>
 <!ELEMENT TArticle (#PCDATA)>
 <!ATTLIST TArticle IDAuts  IDREFS #REQUIRED>
]>
```

FIG. 3.2 – DTD qui modélise les laboratoires attachés à une université

Comme mentionné au début de cette section, le langage de schéma décrit par une DTD correspond à une grammaire d'arbres régulière locale (LTG). Dans la suite nous allons montrer comment une DTD peut être modélisée par une LTG. D'abord nous définissons l'alphabet d'une DTD.

Définition 3.3.2 L'alphabet d'une DTD : Étant donné une DTD d, soit Σ_{ele} l'ensemble des noms d'éléments qui apparaissent dans d, soit Σ_{att} l'ensemble des noms d'attributs qui apparaissent dans d. *L'alphabet d'une DTD* est l'ensemble $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$, où *data* dénote les textes qui apparaissent dans les documents XML, *i.e.*, les types PCDATA et CDATA. □

Dans cette thèse, nous prenons en considération les attributs des DTD et donc, la définition d'une LTG comme une DTD prend en considération les attributs. Cette définition reprend celle de [FL01].

Définition 3.3.3 DTD comme une grammaire d'arbre régulière locale avec les attributs : Soit d une DTD avec l'alphabet $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$. La grammaire d'arbre régulière locale $D = (N, \Sigma_{ele} \cup \{data\}, \Sigma_{att}, P, R_{comp}, R_{opt}, firstEle)$ qui défini d est :

- $N = \Sigma$.
- P est une application qui fait correspondre les éléments de N à leurs définitions, c'est-à-dire, $P(\alpha)$ a la forme $P(\alpha) = e(E|\varepsilon)$
  où $\alpha \in N$, $e \in \Sigma_{ele} \cup \{data\}$, E est une expression régulière sur N et ε représente le mot

vide (remarquer que dans les grammaires locales $\alpha = e$).
- R_{comp} est une application partielle qui fait correspondre $\Sigma_{ele} \cup \{data\}$ à $\mathcal{P}(\Sigma_{att})$ (*i.e.*, l'ensemble des parties de Σ_{att}); si $a \in R_{comp}(e)$ alors a est un attribut obligatoire défini pour e.
- R_{opt} est une application partielle qui fait correspondre $\Sigma_{ele} \cup \{data\}$ à $\mathcal{P}(\Sigma_{att})$; si $a \in R_{opt}(e)$ alors a est un attribut optionnel défini pour e.
- $firstEle \in N$ est l'élément racine. □

Remarquer que $R_{comp} \cup R_{opt}$ représente l'ensemble de tous les attributs définis dans la grammaire locale D. L'exemple ci-dessous montre la DTD de la figure 3.2 comme une grammaire d'arbre régulière locale qui prend en considération les attributs.

Exemple 3.3.1 Étant donné la DTD de la figure 3.2, la grammaire d'arbre régulière locale correspondante est:
- $N = \{Universite, Laboratoire, Nom, Chercheur, Publication, Sujet, Annee, Revue, Titre, TArticle, CId, IdAuts, data\}$
- $\Sigma_{ele} \cup \{data\} = \{Universite, Laboratoire, Nom, Chercheur, Publication, Sujet, Annee, Revue, Titre, TArticle, data\}$
- $\Sigma_{att} = \{CId, IdAuts\}$
- $P(Universite) = Universite(Laboratoire^*)$,
  $P(Laboratoire) = Laboratoire(Nom\ Chercheur^*)$,
  $P(Nom) = Nom(data)$, $P(Chercheur) = Chercheur(Nom\ Titre)$,
  $P(Publication) = Publication(Sujet\ (Annee\ Revue^+)^*)$,
  $P(Sujet) = Sujet(data)$, $P(Annee) = Annee(data)$,
  $P(Revue) = Revue(Nom\ TArticle)$, $P(Titre) = Titre(data)$,
  $P(TArticle) = TArticle(data)$, $P(data) = data(\varepsilon)$.
- $R_{comp}(Chercheur) = \{CId\}$, $R_{opt}(Chercheur) = \emptyset$, $R_{comp}(TArticle) = \{IdAuts\}$,
  $R_{opt}(TArticle) = \emptyset$ (l'image par R_{comp} et R_{opt} des autres non-terminaux est vide).
- $firstEle = Universite$ □

Pour faciliter la construction de validateurs XML fondés sur les DTD, le W3C a préconisé que les expressions régulières utilisées pour définir les modèles de contenus soient non ambiguës [BPSM98]. Cette restriction garantit que la construction des automates d'états finis déterministes pour les expressions régulières est polynomiale [SV02] car les expressions régulières non-ambiguës génèrent des automates d'états finis déterministes sans utiliser un algorithme de déterminisation.

Afin de définir la notion d'expression régulière non-ambiguë, on associe toute expression régulière E à une expression indicée $\overline{E}$ obtenue à partir de E comme suit : le premier symbole de E est indicé par 1, le deuxième symbole de E par 2 et ainsi de suite. Par exemple, étant donné l'expression régulière $E = a(b|c)a$, l'expression indicée correspondante est $\overline{E} = a_1(b_2|c_3)a_4$. Soit $^\#$ la notation utilisée pour le mot obtenu d'un mot w qui appartient à $\overline{E}$ sans ses indices. Les expressions régulières non ambiguës sont définies comme suit.

Définition 3.3.4 Expression régulière non ambiguë [BW98a] : Étant donné une expression régulière E et son expression indicée $\overline{E}$, E est une expression régulière non ambiguë si son expression indicée $\overline{E}$ ne décrit pas deux mots uxv et uyw tels que $x \neq y$ et $x^\# = y^\#$, où u, v et w sont des mots indicés et x et y deux lettres indicées. □

L'exemple ci-dessous montre l'utilisation de la définition 3.3.4 pour déterminer si une expression régulière est ambiguë ou pas.

Exemple 3.3.2 Étant donné l'expression régulière $E = (a|b)^*a$ ($\overline{E} = (a_1|b_2)^*a_3$). Les mots $\alpha = b_2\ a_1\ a_3$ et $\beta = b_2\ a_3$ appartiennent à $L(\overline{E})$. Considérons α comme uxv tel que $u = b_2$, $x = a_1$ et $v = a_3$ et $\beta = uyw$ tel que $u = b_2$, $y = a_3$ et $w = \varepsilon$. Selon la définition 3.3.4, x est

différent de y (*i.e.*, $a_1 \neq a_3$), cependant $x^{\#}$ est égal à $y^{\#}$ (*i.e.*, $a = a$). Donc E est ambiguë. □

Toutes les DTD utilisées pour modéliser des schémas pour des documents XML devraient avoir leurs modèles de contenus non-ambigus.

Dans la suite nous présentons d'autres formalismes de schémas en faisant une comparaison avec les DTD.

3.3.2 Autres langages de spécifications de schémas

Bien que la DTD soit l'un des langages de schémas le plus utilisé [BNB04, Cho02], elle a des limitations, parmi lesquelles on peut énumérer [Age04]:

- Une DTD définit très peu de types de données pour la validation du contenu. En fait, les DTD ont seulement le type texte (*i.e.*, chaîne de caractères).
- Il est impossible d'intégrer d'autres définitions existantes puisque le support des *namespaces*[6] différenciant les langages n'est pas supporté.
- Son pouvoir d'expression est limité au pouvoir des grammaires d'arbres locales.
- Le W3C préconise que le modèle de contenu des éléments soit déterministe, *i.e.*, représenté par une expression régulière non-ambiguë.
- Le langage n'est pas exprimé en XML, ce qui est en contradiction avec la forme même du langage défini.

Par exemple, si on veut définir qu'un élément dans un document XML ne représente que des valeurs entières, cette contrainte ne peut pas être modélisée par une DTD. De même, si on veut modéliser des schémas dans lesquels un même élément aurait des modèles de contenus différents selon le contexte dans lequel il apparaît, cela n'est pas non plus possible en utilisant une DTD. Remarquer que cette dernière limitation existe car les DTD sont des grammaires d'arbres régulières locales.

Du fait des limitations énumérées ci-dessus, d'autres schémas ont été proposés. Dans cette section nous en présentons brièvement deux : XML-Schema et RELAX-NG.

XML-Schema

Le formalisme de schéma XML-Schema est la référence des langages de schémas proposée par le W3C. XML-Schema est une paire de recommandations concernant le typage des documents XML :

- La partie Schema-Structure [TBMM04] propose un langage pour exprimer la structure des éléments XML en fonction d'autres éléments et de types de données;
- La partie Schema-Datatypes [BM04] propose un ensemble de types de données de base (par exemple les dates, entiers) et la manière de les restreindre.

Si nous ne prenons pas en considération certaines caractéristiques du langage XML-Schema (*e.g.*, héritage, dérivation de types, etc.), il est équivalent à la grammaire d'arbre régulière à type unique. Cependant, même si on utilise toutes les caractéristiques de XML-Schema, il n'est pas équivalent à une grammaire d'arbre régulière [MLM01].

Dans un schéma XML-Schema les éléments peuvent comporter des attributs et sont associés à un type de données qui peut être défini comme "type simple" ou "type complexe". Un type simple est un élément ne contenant que du texte, tandis qu'un type complexe est un élément qui peut contenir d'autres éléments ou des attributs. Ces derniers servent à qualifier un élément et ne peuvent être que de type simple. Cependant, il est possible de rendre un attribut optionnel

[6] Domaines de Noms. Il permet de distinguer les noms utilisés dans les documents XML.

ou obligatoire et de lui attribuer une valeur par défaut. C'est ce que l'on appelle les contraintes d'occurrences. XML-Schema offre également la possibilité de constituer des "groupes d'attributs" afin de faciliter à la fois la lisibilité et la maintenance d'un schéma.

Comme caractéristiques propres à XML-Schema nous pouvons énumérer :
- Les schémas XML-Schema sont écrits en utilisant le langage XML.
- Les domaines de noms (*namespaces*) : un schéma est constitué d'un ensemble de définitions de types et de déclarations d'éléments. Pour éviter tout conflit, chacun des noms qui leur sont respectivement attribués appartient à un espace de nom particulier appelé espace de nom cible (*target namespace*). Les domaines de noms permettent donc de fournir un contexte à un vocabulaire, ce qui facilite la création de schémas.
- La dérivation de types : XML-Schema a été conçu de façon à supporter des mécanismes d'héritage. La dérivation par restriction et la dérivation par extension sont les deux techniques mises à disposition et s'appliquent aussi bien aux types de données simples que complexes. Cependant, afin d'éviter toute incohérence, XML-Schema permet de contrôler les dérivations effectuées. Ainsi, l'auteur d'un schéma peut préciser, pour un type simple ou complexe, s'il est dérivable ou non, et si oui, quel type de dérivation peut lui être affecté.
- La dérivation par restriction : cette technique permet de créer de nouveaux types simples, à partir de la bibliothèque de types simples intégrée à XML-Schema. Ainsi, il est possible de restreindre l'ensemble des valeurs légales d'un type de base. Parmi les différentes façons qui existent, les notions "enumeration" et "pattern" sont particulièrement utiles. La première permet d'énumérer l'ensemble des valeurs possibles pouvant être attribuées à un type simple, l'autre permet de définir des expressions régulières. Dans le cas de la restriction de types complexes, les valeurs représentées par le nouveau type sont un sous-ensemble des valeurs du type de base.
- La dérivation par extension : l'extension de types simples consiste en fait, à créer des types complexes dont le contenu est un type simple pouvant porter des attributs. Pour ce qui est de l'extension de types complexes, elle permet de rajouter des éléments ou des attributs au modèle complexe de base.

La figure 3.3 présente la traduction de la DTD de la figure 3.2 dans la forme XML-Schema. On notera que le document de la figure 2.1 est conforme au schéma de la figure 3.3. D'autre part, on peut voir que le schéma défini par XML-Schéma est plus verbeux que celui défini par la DTD.

Comme les schémas décrits en XML-Schéma sont des grammaires d'arbres régulières à type unique et les DTD sont des grammaires d'arbres régulières locales, des schémas DTD peuvent être modélisés par des schémas XML-Schema, cependant le contraire n'est pas possible.

RELAX-NG

Le langage RELAX-NG [CM01] résulte de la fusion de deux langages de schéma : TREX (Tree Regular Expressions for XML) et RELAX (REgular LAnguage description for XML). Ce langage permet de définir des schémas équivalents à des grammaires d'arbres régulières.

Les schémas RELAX-NG sont écrits en utilisant le langage XML. RELAX-NG représente les règles de production $X \rightarrow \mathbf{a}\ (r)$ d'une RTG par des éléments *define*. L'attribut *name* d'un élément *define* définit le membre gauche d'une règle de production (*i.e.*, X) de la grammaire et les éléments fils représentent le membre droit de la règle (*i.e.*, $\mathbf{a}\ (r)$). Le symbole terminal et le modèle de contenu qui apparaissent dans le membre droit de la règle de production sont représentés par l'élément fils *element* et par les fils de l'élément *element*, respectivement.

RELAX-NG a deux notions que les deux langages de schémas présentés précédemment n'ont

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Annee">
    <xs:complexType mixed="true" />
  </xs:element>
  <xs:element name="Chercheur">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Nom" />  <xs:element ref="Titre" />
      </xs:sequence>
      <xs:attribute name="CId" type="xs:ID" use="required" />
    </xs:complexType>
  </xs:element>
  <xs:element name="Laboratoire">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Nom" />
        <xs:element ref="Chercheur" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Nom">
    <xs:complexType mixed="true" />
  </xs:element>
  <xs:element name="Publication">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Sujet" />
        <xs:sequence>
          <xs:element ref="Annee" /> <xs:element ref="Revue" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Revue">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Nom" /> <xs:element ref="TArticle" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Sujet">
    <xs:complexType mixed="true" />
  </xs:element>
  <xs:element name="TArticle">
    <xs:complexType mixed="true">
      <xs:attribute name="IDAut's"type="xs:IDREFS" use="required" />
    </xs:complexType>
  </xs:element>
  <xs:element name="Titre">
    <xs:complexType mixed="true" />
  </xs:element>
  <xs:element name="Universite">
    <xs:complexType>
      <xs:sequence>
         <xs:element ref="Laboratoire" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

FIG. 3.3 – Un schéma XML-Schema qui modélise les laboratoires attachés à une université.

pas. D'abord les définitions des attributs sont plus riches : on peut définir des choix entre attributs. Par exemple, si le nom d'un client est caractérisé soit par son prénom et nom de famille soit simplement par son nom de famille, RELAX-NG permet de définir ce choix dans le schéma et ainsi, les documents peuvent avoir les deux types de caractérisation de noms de clients. La partie suivante d'un schéma RELAX-NG présente cette notion.

```
<element name="client">
   <choice>  <attribute name="nom">   <text/>  </attribute>
      <group>  <attribute name="prenom"> <text/> </attribute>
               <attribute name="nom"> <text/> </attribute>
      </group>
   </choice>
</element>
```

Ainsi, l'élément *client* dans un document XML conforme au morceau ci-dessus peut avoir :
<client nom="Dupont"/> **et/ou** <client prenom="Jean" nom="Dupont" />

On peut aussi définir qu'une valeur sera modélisée par des attributs ou par des éléments. Par exemple, on peut remplacer le morceau du schéma ci-dessus par :

```
<element name="client">
   <choice>  <element name="nom">   <text/>  </element>
      <group>  <attribute name="prenom"> <text/> </attribute>
               <attribute name="nom"> <text/> </attribute>
      </group>
   </choice>
</element>
```

Ainsi, un document XML conforme à un schéma RELAX-NG peut avoir des noms de clients sous la forme d'attributs ou d'élément :

<client prenom="Jean" nom="Dupont" />

et/ou

<client> <nom> Jean Dupont </nom> </client>

La deuxième notion originale de RELAX-NG est la notion d'intercalage (*interleave*) qui permet à des éléments fils d'apparaître dans n'importe quel ordre. L'exemple suivant permet à l'élément *client* de contenir les éléments *nom* et *prenom* dans un ordre quelconque :

```
<element name="client">
  <interleave> <element name="nom"> <text/> </element>
               <element name="prenom"> <text/> </element>
  </interleave>
</element>
```

Dans ce cas, un document XML valide peut avoir dans son contenu :
<client> <prenom> Jean </prenom> <nom> Dupont </nom> </client>

et/ou

<client> <nom> Dupont </nom> <prenom> Jean </prenom> </client>

La figure 3.4 présente un exemple de schéma RELAX-NG qui modélise la même application que celle modélisée par la DTD de la figure 3.2.

```
<?xml version="1.0"?>
<grammar ns="" xmlns="http://relaxng.org/ns/structure/1.0"
         datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
    <start>
        <choice>
            <ref name="TArticle"/>
            <ref name="Sujet"/>
            <ref name="Chercheur"/>
            <ref name="Laboratoire"/>
            <element name="Publication">
                <ref name="Sujet"/>
                <zeroOrMore>
                    <ref name="Annee"/>
                    <oneOrMore>
                        <ref name="Revue"/>
                    </oneOrMore>
                </zeroOrMore>
            </element>
            <ref name="Nom"/>
            <ref name="Annee"/>
            <ref name="Revue"/>
            <element name="Universite">
                <zeroOrMore>
                    <ref name="Laboratoire"/>
                </zeroOrMore>
            </element>
            <ref name="Titre"/>
        </choice>
    </start>
    <define name="Nom">
        <element name="Nom">
            <text/>
        </element>
    </define>
    <define name="Chercheur">
        <element name="Chercheur">
            <attribute name="CId">
                <data type="ID"/>
            </attribute>
            <ref name="Nom"/>
            <ref name="Titre"/>
        </element>
    </define>
    <define name="Titre">
        <element name="Titre">
            <text/>
        </element>
    </define>
    <define name="TArticle">
        <element name="TArticle">
            <attribute name="IDAuts">
                <data type="IDREFS"/>
            </attribute>
            <text/>
        </element>
    </define>
    <define name="Revue">
        <element name="Revue">
            <ref name="Nom"/>
            <ref name="TArticle"/>
        </element>
    </define>
    <define name="Sujet">
        <element name="Sujet">
            <text/>
        </element>
    </define>
    <define name="Annee">
        <element name="Annee">
            <text/>
        </element>
    </define>
    <define name="Laboratoire">
        <element name="Laboratoire">
            <ref name="Nom"/>
            <zeroOrMore>
                <ref name="Chercheur"/>
            </zeroOrMore>
        </element>
    </define>
</grammar>
```

FIG. 3.4 – Un schéma RELAX-NG qui modélise les laboratoires attachés à une université.

3.3.3 Discussion

L'expressivité des langages de schémas présentés dans cette section est relative aux grammaires d'arbres auxquelles ils correspondent :

- La DTD : il n'est pas possible de définir un élément plus qu'une fois dans une DTD. Cela évite la concurrence entre symboles non-terminaux et correspond donc aux grammaires locales.
- Le XML-Schema : il est possible de définir plus d'un modèle de contenu pour un élément, cependant les éléments ayant des modèles de contenus différents ne peuvent pas faire partie de l'ensemble du modèle de contenu d'un troisième élément : cela correspond aux grammaires à type unique.
- Le RELAX-NG : il correspond aux grammaires d'arbres régulières, ainsi il n'existe pas de restriction pour la construction des modèles de contenus et pour les définitions des éléments.

Même si le formalisme DTD est moins expressif que les formalismes XML-Schema et RELAX NG, il est toujours utilisé dans de très nombreuses applications [Age04]. Dans [BNB04], les auteurs ont fait une comparaison de l'utilisation des DTD et des XML-Schema. Ils ont conclu que la majorité des caractéristiques exclusives au langage XML-Schema ne sont pas très utilisées, particulièrement celles qui sont liés au modèle de données orienté objet comme la dérivation de types complexes par extension. En général, l'expressivité des schémas écrits en XML-Schema dans les applications est celui des DTD, cela veut dire que ces schémas pourraient être définis par des DTD.

RELAX NG est le formalisme le plus expressif et il est plutôt récent. Il n'existe pas beaucoup d'outils pour travailler avec RELAX-NG [Age04]. Par exemple, la majorité des éditeurs d'EAI (plates-formes d'intégration d'applications) et des outils graphiques utilise les formalismes XML-Schema et DTD.

Deuxième partie

État de l'art

Chapitre 4

Différentes approches de validation de documents XML par automates

Dans le domaine des bases de données, la majorité des documents XML est associée à des schémas [Suc01]. En effet, la connaissance de la structure du document peut rendre sa manipulation plus efficace. Ainsi, la validation des documents XML par rapport à leurs schémas joue un rôle important dans le cadre de XML.

Le terme valide est souvent utilisé pour définir un document qui respecte un schéma donné. Le processus de vérification de la validité d'un document par rapport à son schéma est appelé validation. Plusieurs approches pour la validation ont déjà été proposées [Chi00, CR04, FL01, HM02, IDD04, MLMK04, MH03, SV02]. Dans ce chapitre nous nous intéressons à certaines d'entre elles.

Nous parlerons aussi des validations incrémentales, c'est-à-dire les validations qui prennent en considération une partie d'un document et, éventuellement, des informations auxiliaires obtenues par des validations précédentes. Ce type de validation est appliqué dans le cadre des mises à jour des documents XML. Remarquer que les logiques (*e.g.*, logique du premier ordre, logique du second ordre) peuvent aussi être utilisées comme des représentations des schémas [BCT04, CDL99, CNT04, CM03, DL03, KSS03], cependant nous ne les aborderons pas ici car nous présenterons plutôt les techniques directement liées à nos travaux.

4.1 La validation par automates d'états finis

Les automates d'états finis dans la validation des documents XML sont utilisés quand les données sont traitées comme un flot continu[1] et sous des contraintes de mémoire centrale. Dans ce contexte, en général, la validation des attributs des documents est ignorée [CR04].

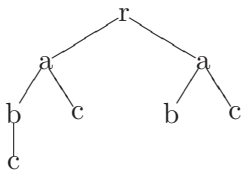
Le processeur de lecture en continu *voit* un document XML comme une séquence d'ouverture et fermeture de balises dans l'ordre de lecture. Le document est donc traité comme une chaîne de caractères. Soit Σ un alphabet fini composé des étiquettes des balises d'un document XML. Étant donné $a \in \Sigma$, a représente l'ouverture d'une balise et $\overline{a}$ la fermeture de la balise a. L'alphabet $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$ représente les symboles de fermeture de balises. Par exemple, le document XML de la figure 4.1 est représenté par la chaîne $rabc\overline{c}\overline{b}bc\overline{c}\overline{a}ab\overline{b}bc\overline{c}\overline{a}r\overline{r}$.

Dans [CR04, SV02], la représentation sous forme de chaîne de caractères d'un arbre t, noté $[t]$, est définie de la façon suivante : (i) si t est un arbre avec un seul nœud étiqueté a, alors

[1] Nous allons utiliser le terme *flot continu* ou *lecture en continu de données* pour *streaming*.

```
<r>
  <a>
    <b> <c></c> </b>
    <c></c>
  </a>
  <a>
    <b></b>
    <c></c>
  </a>
</r>
```

(a)



(b)

FIG. 4.1 – Représentation d'un document XML (a) comme une séquence de balise et (b) comme un arbre.

$[t] = a\overline{a}$ et (ii) si t a un nœud racine a et des sous-arbres $t_1 \dots t_k$, alors $[t] = a[t_1] \dots [t_k]\overline{a}$. Remarquons que pour chaque arbre XML t, $[t]$ est une chaîne de caractères bien formée qui correspond au parcours en profondeur de t. L'ensemble d'arbres qui respectent un schéma d est noté $L(d)$. Le langage $\mathcal{L}(d)$ (sur $\Sigma \cup \overline{\Sigma}$) représente les documents de $L(d)$ sous forme de chaînes de caractères, *i.e.*, $\{[t] \mid t \in L(d)\}$. Pour reconnaître le langage défini par un schéma d dans le contexte de flot continu, les automates d'états finis peuvent être utilisés.

Pour la validation d'un document XML vu comme un flot continu de données, et en prenant en considération les contraintes de mémoire centrale, l'utilisation des automates d'états finis est alors envisagée [CR04, SV02]. Cela veut dire que les schémas doivent être représentés par des expressions régulières. Cette restriction peut poser des problèmes si le processus de validation vérifie aussi la bonne formation des documents, surtout l'imbrication et l'ouverture et fermeture des balises. Ainsi, pour identifier les classes de schémas pour lesquelles on peut utiliser les automates d'états finis pour valider les documents sous forme de flot continu, une caractérisation des schémas est proposée dans [SV02] :

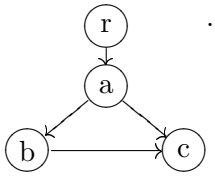
1. Un schéma est *fortement reconnaissable* s'il est possible de vérifier la validation et la bonne formation des documents associés en utilisant un automate d'états finis.
2. Un schéma est *reconnaissable* s'il est possible de vérifier la validation des documents associés en utilisant un automate d'états finis. Les documents doivent être bien-formés.
3. Un schéma est *non-reconnaissable* s'il n'est pas possible de vérifier la validation des documents associés en utilisant un automate d'états finis.

Avant de présenter des exemples nous allons montrer comment les schémas sont classés.

La caractérisation de chaque classe est basée sur le graphe de dépendance des schémas. Soit la grammaire d'arbre $D = (N, \Sigma, P, I)$ (définition 3.1.3). Un graphe de dépendance $G_D = (X, U)$ de D est construit comme suit [SV02] : l'ensemble de nœuds X est l'ensemble N, *i.e.*, $X = N$ et, chaque règle de production $a \to \mathbf{a}\ (R_a) \in P$ engendre, pour chaque b qui apparaît dans l'expression régulière R_a, un arc de a vers b.

Exemple 4.1.1 Étant donné les règles de production de la grammaire d'arbre régulière (locale) D

$r \to \mathbf{r}\ (a^*)$
$a \to \mathbf{a}\ (b\ c)$
$b \to \mathbf{b}\ (c?)$
$c \to \mathbf{c}\ (\varepsilon)$,

le graphe de dépendance correspondant est : 

. □

En utilisant le graphe de dépendance d'un schéma il est possible de démontrer [SV02] :
- Les schémas dont les graphes de dépendance n'ont pas de cycles (des graphes acycliques) sont des schémas fortement reconnaissables.
- Les schémas dont les graphes de dépendance ont des cycles sont dans les classes des schémas reconnaissables ou non-reconnaissables. La classification dépend du niveau de la récursivité et pour la présenter nous avons besoin de considérer les rapports entre les nœuds dans un graphe de dépendance et leurs cycles.

Soient une grammaire D et son graphe de dépendance $G_D = (X, U)$. Deux types (symboles non-terminaux) a et b de D sont mutuellement récursifs si les nœuds correspondants dans G_D appartiennent à un même cycle. De plus, un type c est mutuellement récursif s'il est récursif avec lui-même (représente les cas ayant la forme $c \rightarrow \mathbf{c}(c)$). Le schéma D est récursif complet si tous les nœuds de G_D depuis lesquels les nœuds récursifs sont atteints sont mutuellement récursifs. Remarquer que deux symboles a et b sont mutuellement récursifs si b apparaît dans l'expression régulière de la règle de production de a et vice-versa.

Exemple 4.1.2 Étant donné un schéma D et les règles de production $r \rightarrow \mathbf{r}\ (a)$ et $a \rightarrow \mathbf{a}\ (a?)$ de D, D est récursif, cependant il n'est pas récursif complet car le nœud r a un arc vers le nœud a qui est (mutuellement) récursif, et r n'est pas récursif. Cependant, un schéma D' avec les règles $r \rightarrow \mathbf{r}\ (a)$ et $a \rightarrow \mathbf{a}\ (r)$ est récursif complet. □

La caractérisation des schémas récursifs reconnaissables est faite par le lemme 4.1.1 ci-dessous. Intuitivement, ce lemme caractérise qu'un document valide par rapport à un schéma récursif et qui est reconnu par un automate d'états finis doit donner les informations pour que l'automate l'accepte sans faire *back tracking*. En d'autres termes, lorsque l'automate trouve une balise de fermeture qui représente un élément récursif, il n'a pas d'information précédent pour "savoir" à quel contexte l'élément fermé appartient. Par exemple, étant donné $a \rightarrow \mathbf{a}(ab|ca)?$, lorsque la balise $\overline{a}$ est trouvé, l'automate d'états finis ne "sait" pas si la dernière balise lue a été c ou a. Si la balise lue est c, alors la prochaine ne peut pas être b.

Lemme 4.1.1 [SV02] *Soit D un schéma reconnaissable. Soit R_a le membre droit de la règle de production d'une étiquette a de D. Alors le fait suivant est respecté, où α, β, u, v, $w \in \Sigma^*$ et x, y, $z \in \Sigma$:*
Soient k un entier positif et x_i, z_i, $1 \leq i \leq k$ des symboles mutuellement récursifs de D (x_i et z_i peuvent représenter le même symbole). Si $\alpha x_1 \beta \in R_{z_1}$, $\alpha' x_k \beta' \in R_{z_1}$ et $u_i x_{(i-1)} v_i x_i w_i \in R_{z_i}$, alors $\alpha x_1 v_2 x_2 \ldots v_k x_k \beta' \in R_{z_1}$.

La condition imposée pour les schémas reconnaissables peut être simplifiée comme suit (dans le cas de $k = 1$) : soient x et z des symboles mutuellement récursifs dans D (par exemple, $x \rightarrow \mathbf{x}\ (z)$ et $z \rightarrow \mathbf{z}\ (x)$). Si les mots $\alpha x \beta$ et $\alpha' x \beta'$ appartiennent au langage régulier de l'expression régulière R_z de la règle de production dont z est le membre gauche, alors le $\alpha x \beta'$ doit aussi être dans $L(R_z)$.

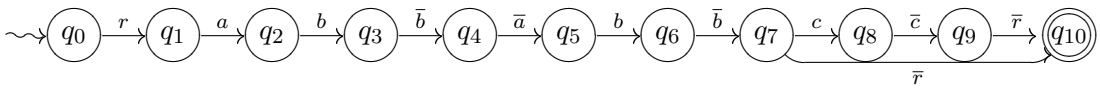
Exemple 4.1.3 Étant donné les règles d'un schéma D : $add \rightarrow$ **add** $(add\ ville|rue\ add)?$, $ville \rightarrow$ **ville** $(data)$ et $rue \rightarrow$ **rue** $(data)$, où le symbole add est mutuellement récursif avec lui-même. D n'est pas reconnaissable car il ne respecte pas la condition établies par le lemme 4.1.1. Soient $k = 1$ et $R_{add} = (add\ ville|rue\ add)?$ l'expression régulière d'élément add dont le langage est $\{\varepsilon, add\ ville, rue\ add\}$. Selon le lemme 4.1.1, pour que D soit reconnaissable, R_{add} doit aussi décrire le mot $rue\ add\ ville$. Comme R_{add} ne le décrit pas alors D n'est pas reconnaissable. □

Ainsi, pour les documents qui ont des schémas non-récursifs, on peut utiliser les automates d'états finis pour vérifier s'ils sont bien-formés et valides. Tandis que pour les documents qui ont des schémas récursifs, les automates d'états finis peuvent être utilisés seulement pour une classe de schémas qui respecte le lemme 4.1.1.

4.1.1 Construction des automates d'états finis correspondant à un schéma

Dans [CR04], l'approche utilisée pour transformer un schéma en un automate d'états finis est fondée sur une grammaire spéciale appelée *XML-grammar* [BB00]. Une *XML-grammar* est une grammaire ayant les deux alphabets Σ et $\overline{\Sigma}$. Par exemple, étant donné les règles d'un schéma $D : r \rightarrow \mathbf{r}\ (a\ b\ c?)$, $a \rightarrow \mathbf{a}\ (b)$, $b \rightarrow \mathbf{b}\ (\varepsilon)$ et $c \rightarrow \mathbf{c}\ (\varepsilon)$, D transformé en *XML-grammar*, notée D_{XG}, est $r \rightarrow r\ a\ b\ c?\ \overline{r}$, $a \rightarrow a\ b\ \overline{a}$, $b \rightarrow\ b\ \varepsilon\ \overline{b}$ et $c \rightarrow c\ \varepsilon\ \overline{c}$.

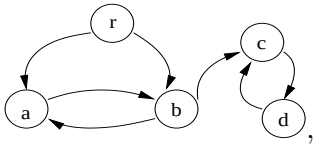
Le schéma utilisé dans l'approche proposée dans [CR04] est équivalent à une grammaire d'arbre régulière locale (*i.e.*, une DTD) non-récursive (*i.e.*, les schémas sont fortement reconnaissables). Les auteurs proposent d'abord la transformation des DTD en *XML-grammar* et ensuite, les *XML-grammar* sont transformées en expressions régulières. La transformation est guidée par la règle qui contient le symbole initial de la grammaire. Chaque symbole e dans l'expression régulière de la règle qui contient le symbole initial est remplacé par l'expression régulière de la règle de production dont e est le membre gauche. Le processus continue récursivement jusqu'à ne plus avoir de règle de production à utiliser. Par exemple, la grammaire D_{XG} ci-dessus est transformée dans l'expression régulière $E = r\ a\ b\ \overline{b}\ \overline{a}\ b\ \overline{b}\ (c\ \overline{c})?\ \overline{r}$. E est transformée en un automate d'états finis (l'algorithme de transformation utilisé est l'algorithme de Glushkov, introduit dans la section 7.2). Par exemple, E donnera l'automate d'états finis :



Dans [SV02] deux algorithmes de transformation de schémas en automates d'états finis sont proposés. Le premier qui est similaire à l'algorithme proposé dans [CR04] traite les schémas non-récursifs. Le deuxième traite les schémas récursifs.

L'algorithme qui traite les schémas récursifs s'appuie sur le graphe de dépendance. Soient D un schéma et $G_D = (X, U)$ le graphe de dépendance de D. D'abord, les composantes fortement connexes de G_D sont construites et ensuite, il est établi une hiérarchie entre les composantes construites. Nous allons appeler les composantes fortement connexes des classes (les classes sont des ensembles des nœuds dans X). La hiérarchie entre les classes est établie par un ordre partiel $\prec$: soit A et B deux classes, $A \prec B$ s'il existe un nœud $a \in A$ et un nœud $b \in B$ tels qu'il existe un arc depuis a vers b dans U. Si on voit la hiérarchie des classes comme un arbre t_H, cet arbre aura une seule classe racine (c'est la classe contenant le symbole initial de D). De plus, les classes dans les feuilles sont les classes pour lesquelles les symboles sont membres gauches d'une règle de production ayant une expression régulière vide ou une expression régulière qui contient des symboles de la même classe. La classe dans la racine est appelée *classe minimale* et les classes dans les feuilles sont appelées *classes maximales*.

Exemple 4.1.4 Étant donné les règles de production de la grammaire $D : r \rightarrow \mathbf{r}\ (a\ b)$, $a \rightarrow \mathbf{a}\ (b)$, $b \rightarrow \mathbf{b}\ (a\ c)$, $c \rightarrow \mathbf{c}\ (d)$ et $d \rightarrow \mathbf{d}\ (c)$, et son graphe de dépendance correspondant G_D :

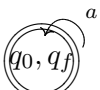


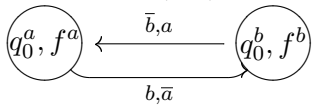
, les classes (composantes fortement connexes) sont $R = \{r\}$, $A = \{a, b\}$ et $C = \{c, d\}$ et la hiérarchie de classes est $R \prec A \prec C$. La classe minimale est R et la classe maximale est C. □

La construction de l'automate pour un schéma récursif commence par les classes maximales. Pour chaque élément e dans les classes maximales, un automate d'états finis est construit pour l'expression régulière de la règle de production de e. Après la construction des automates pour tous les éléments d'une classe C, les automates sont fusionnés : pour chaque transition $\delta(q, b) = q'$ de l'automate M_a construit à partir de l'expression régulière de la règle de a tel que $b \in C$. Le nouvel automate aura un arc de l'état q de M_a vers l'état initial q_0 de M_b étiqueté b et des arcs des états finaux de M_b vers l'état q' de M_a étiqueté $\overline{b}$. Lorsque toutes les classes sont représentées par des automates d'états finis, ces automates sont fusionnées comme mentionné ci-dessus. Ce processus se répète en remontant la hiérarchie jusqu'à la classe racine. L'automate résultant aura un état initial : il est ajouté avec le symbole r qui représente la racine. À partir de tous les états finaux, un arc est inséré vers le nouvel état final étiqueté $\overline{r}$.

Ci-dessous, nous présentons un exemple qui illustre la construction d'un automate d'états finis pour un schéma récursif.

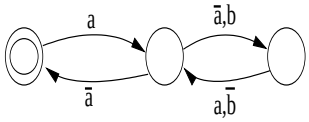
Exemple 4.1.5 Soit le schéma D (récursif complet) avec les règles de production suivantes : $a \rightarrow \mathbf{b}(b^*)$ et $b \rightarrow \mathbf{a}(a^*)$ ayant a comme symbole initial. Soit $G_D = (X, U)$ le graphe de dépendance de D. La seule classe construite à partir de G_D est $C = \{a, b\}$. Les automates M_a et M_b sont construits à partir des expressions régulières qui apparaissent dans les règles de productions de a et de b, respectivement : M_a 

et M_b 

, où les états q_0 et q_f représentent les états initiaux et finaux des automates, respectivement. L'automate pour la classe C est construit en utilisant les états des automates M_a et M_b et en ajoutant des transitions $\delta(q, b) = q_0$ et $\delta(q_f, \overline{b}) = q'$ pour chaque transition $\delta(q, b) = q'$ dans les automates M_a et M_b, ce qui produit l'automate M_C suivant : 

.

Enfin, en prenant en considération que le symbole a est le symbole initial de D, l'automate d'états finis M_D qui accepte les documents qui respectent D est construit :



□

L'automate d'états finis construit dans l'exemple 4.1.5 accepte seulement tous les documents bien-formés qui respectent le schéma D.

La caractérisation des schémas récursifs par rapport aux automates d'états finis est encore un problème ouvert : il existe des schémas récursifs pour lesquels la méthode mentionnée ci-dessus construit des automates qui acceptent plus de documents que les schémas n'en décrivent.

Par exemple, soit D un schéma récursif ayant les règles de production $r \rightarrow aa$ et $a \rightarrow a?$, l'automate M résultant de l'application des pas ci-dessus (figure 4.2) accepte non seulement tous les documents qui respectent D mais aussi des documents ayant la forme de la chaîne $raa\overline{a}a\overline{aar}$.

Les documents traités sous forme de chaîne de caractères modélisés par des schémas récursifs peuvent être validés par d'autres types d'automates (par exemple, les automates à piles). Cependant, la contrainte de mémoire centrale doit être levée. Dans [SV02], il est proposé d'utiliser les automates à piles déterministes et dans [CR04], les auteurs utilisent les automates *one-counter* (1-CFSA).

Un *1-CFSA* est un quintuplet $M = (Q, \Sigma, H, q_0, F)$, où Q est l'ensemble fini d'états, $q_o \in Q$ est l'état initial, $F \subset Q$ est l'ensemble d'états finals, Σ est l'alphabet de M et H est un sous-ensemble fini de $Q \times (\Sigma \cup \varepsilon) \times \mathbb{Z}^+ \times \{-1, 0, 1\} \times Q$. Un *1-CFSA* est un automate d'états finis avec

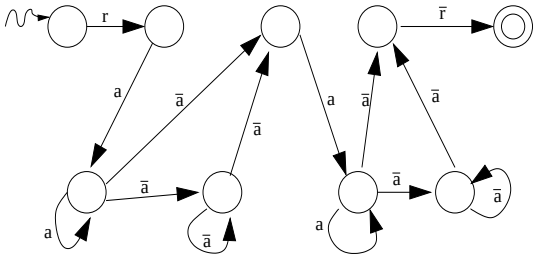
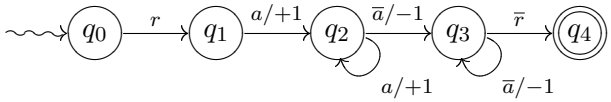


FIG. 4.2 – L'automate M construit à partir des règles de production $r \rightarrow aa$ et $a \rightarrow a?$.

un compteur c qui ne peut avoir que des valeurs positives ($c \geq 0$). L'automate peut tester si le compteur est égal à zéro (0). Une transition peut incrémenter c de 1 (si $c \geq 0$) ou décrémenter c de 1 (si $c > 0$). L'automate accepte un mot s'il commence par l'état initial et termine dans l'état final, le compteur égal à zéro et l'entrée entièrement lue. Les langages acceptés par un *1-CFSA* sont des langages du type $a^n b^n$ (avec $n \geq 0$). Ainsi le schéma $d : r \rightarrow a\ a \rightarrow a|\varepsilon$ transformé en $ra^n\overline{a}^n\overline{r}$ est représenté par l'automate *1-CFSA* : 

Un *1-CFSA* n'accepte pas les mots d'un langage du type $a^n \ldots b^m c^m \ldots d^n$ car le compteur est incrémenté, par exemple, par les a et b et décrémenté par les c et d. Si le nombre final de a et b est égal au nombre final de c et d, le compteur aura 0 cependant le mot peut ne pas appartenir au langage décrit par l'automate. Par exemple, étant donné l'expression régulière $E = a^n b^m c^m a^n$ ($0 < n$ et $0 < m$) et le mot $w = aaabbcaaaa$, un automate *1-CFSA* accepte w car le compteur est incrémenté jusqu'à 5 (3 a et 2 b), ensuite décrémenté en 5 (1 c et 4 a) et le mot w est entièrement lu.

4.1.2 Discussion

Dans cette section nous avons présenté les validations des documents XML sous la forme de flot continu de données. Ce type de validation est utilisé pour la validation des documents XML à la volée : le processus de validation est déclenché dès que les données du document arrivent, c'est-à-dire, l'application n'attend pas que le document arrive entier pour initialiser la validation. Comme le montrent les travaux de [SV02] et de [CR04], ce type de validation peut être difficile à exécuter s'il existe des contraintes de mémoire centrale, principalement, si les documents à valider ont une taille très importante (de l'ordre de gigaoctets). Les automates d'états finis n'utilisent aucune structure auxiliaire pour accepter un mot (document XML), ils sont donc de bonnes solutions pour ce contexte.

Bien que les automates d'états finis soient idéaux dans ce contexte, ils ne peuvent pas être utilisés pour tous les types de schémas. Cette restriction est due du fait que ce type d'automate n'accepte que des mots décrits par des expressions régulières. La transformation d'une grammaire d'arbres (récursive) en une grammaire sur mots peut engendrer des grammaires qui ne sont pas régulières et dans ce cas, les mots ne peuvent pas être acceptés par des automates d'états finis. Pour résoudre cela, des automates à pile peuvent être utilisés mais il est alors nécessaire de relaxer les contraintes de mémoire centrale. Toutefois, la caractérisation des DTD (fortement) reconnaissables proposées dans les travaux [SV02, CR04] vérifient la majorité des DTD trouvées dans des exemples réels [Cho02].

4.2 La validation par des automates d'arbre

Dans la majorité des travaux concernant le traitement (*e.g.*, validation, correction, interrogation, entre autres) des documents XML, les documents sont vus comme des arbres d'arité non-bornée ou des arbres binaires [ABS00, MLMK04, Nev02a, Suc01, Via01]. Dans cette section nous présentons des travaux qui utilisent les automates d'arbre pour valider des documents XML vus comme des arbres.

Les arbres et les automates d'arbres sont étudiés depuis longtemps, cependant les liens entre les automates d'arbre et les arbres dans le contexte de SGML (le prédécesseur du langage XML) ont été faits par Brüggeman-Klein, Murata et Wood [BW98b, Mur95, Mur98]. Les concepts et les formalismes proposés par ces auteurs ont été la référence de plusieurs travaux sur les arbres (XML) d'arité non-bornée.

Les automates d'arbre d'arité bornée :

L'utilisation des arbres d'arité bornée et des automates d'arbre d'arité bornée dans le cadre de la validation des documents XML s'appuie sur les arbres d'arité binaire [Nev02b, PV03, Suc02a]. Dans ce cas, il est nécessaire de lire les documents XML, qui sont naturellement des arbres d'arité non-bornée, de façon à les transformer en arbres binaires. Par ailleurs, si on utilise des formalismes de schémas qui s'appuient sur des expressions régulières (*e.g.*, DTD, XML-Schema, RELAX NG, etc.), il faut transformer les schémas en automates d'arbre binaire. Dans la suite, nous introduisons les automates d'arbre binaire en nous appuyant sur les arbres d'arité non-bornée.

Étant donné un arbre d'arité non-bornée t étiqueté en Σ, un arbre binaire $enc(t)$ sur l'alphabet $\Sigma_{\#} = \Sigma \cup \{\#\}$, où $\# \notin \Sigma$, est construit. La fonction enc code t dans un arbre binaire t_b étiqueté sur $\Sigma_{\#}$ de la façon suivante :

- La racine de t est la racine de t_b.
- Le premier nœud fils n' d'un nœud n dans un arbre t reste toujours le premier fils de n dans $enc(t)$, c'est-à-dire le fils gauche.
- Les autres nœuds fils de n deviennent des descendants à droite de n'.
- S'il existe un nœud fils à droite et il n'existe pas un nœud fils à gauche, un nœud étiqueté # est inséré à gauche.
- S'il n'existe qu'un nœud fils à gauche, un nœud étiqueté # est inséré comme un fils à droite.

De même, la fonction dec est définie comme suit : étant donné un abre d'arité non-bornée t, $t = dec(enc(t))$. La figure 4.3 montre un arbre d'arité non-bornée et sa version binaire complète.

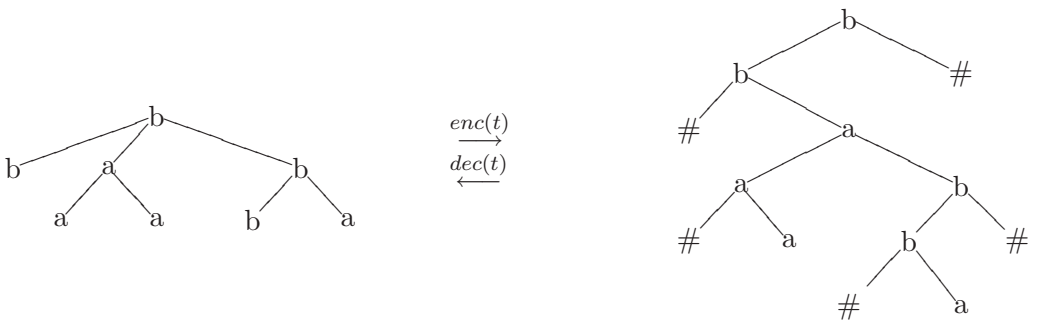


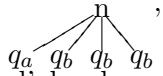
FIG. 4.3 – Un arbre d'arité non borné et sa version codée en binaire complète.

Un document XML vu comme un arbre binaire peut être validé par un automate d'arbre binaire (ascendant ou descendant). Un automate d'arbre binaire est un quintuplet [Nev02b] $A = (Q, \Sigma, \Delta, I, F)$, où :
(i) Q est l'ensemble d'états,
(ii) Σ est l'alphabet de A,
(iii) Δ est une fonction $Q \times Q \times \Sigma \rightarrow Q$ qui fait correspondre à paire d'états à un état,
(iv) I est l'ensemble d'états initiaux tel que $I \subseteq Q$, et
(v) F est l'ensemble d'états finaux tel que $F \subseteq Q$.
Soit t un arbre binaire. L'exécution de A sur t est une correspondance $\lambda : dom(t) \rightarrow Q$ tel que :
- Pour chaque nœud feuille n de t, $\lambda(n) \in I$.
- Pour tous les nœuds internes de t, $\lambda(u) \in \Delta(\lambda(u1), \lambda(u2), u)$ tel que ui est le i-ème fils de u.

Proposition 4.2.1 [Suc02a]

1. *Pour tous les automates d'arbre d'arité non-bornée B, il existe un automate d'arbre binaire A tel que $L(A) = \{enc(t) \mid t \in L(B)\}$.*
2. *Pour tous les automates d'arbre binaire A, il existe un automate d'arbre d'arité non-bornée B tel que $L(B) = \{dec(t) \mid t \in L(A)\}$.*

Les automates d'arbre d'arité non-bornée :

Plusieurs travaux considèrent la validation par des automates d'arbre d'arité non-bornée [BDH04c, BGMT02, Chi00, FL01, MN03, MLMK04]. Les automates d'arbres utilisés pour accepter les arbres d'arité non-bornée (comme ceux présentés dans les sections 3.1.3 et 3.2) sont des mécanismes pour accepter les arbres dérivés de grammaires d'arbres (régulières). En d'autres termes, les schémas sont représentés par des automates d'arbres et leurs langages représentent les arbres XML valides. Les règles de transition d'un automate d'arbre (ou les règles de production de la grammaire) ont des expressions régulières comme des membres droits. Ainsi, des automates d'états finis sont toujours associés aux automates d'arbres pour reconnaître les langages décrits par les expressions régulières. Par exemple, étant donné une règle de transition $\delta(q_a\ q_b^*, c) \rightarrow q_c$ d'un automate d'arbre A et le morceau d'un arbre XML t : 

, où n est un nœud étiqueté c, A associe l'état q_c au nœud n de t en construisant, d'abord, un mot à partir de fils de n. La construction du mot est faite en concaténant les fils de n de gauche à droite, ce qui produit le mot $w = q_a\ q_b\ q_b\ q_b$. Ensuite, A vérifie si $w \in L(q_a\ q_b^*)$, pour cela, un automate d'états finis M construit à partir de $q_a\ q_b^*$ est utilisé. Si M accepte w alors A associe l'état q_c au nœud étiqueté c de t.

Les grammaires d'arbre régulières locales produisent des automates d'arbre déterministes alors que les grammaire d'arbre régulière (non locale) produisent des automates d'arbre non-déterministes [MLMK04]. La validation des arbres qui respectent des schémas plus généraux peut être plus complexe.

Exemple 4.2.1 Soient l'arbre t de la figure 4.4 et l'automate d'arbre $M_2 = (\Sigma, D, Q, \Delta, F)$ (qui représente une grammaire d'arbre régulière) :
$\Sigma = \{section, paragraph\}$
$D = \{\#PCDATA\}$
$Q = \{q_1, q_2, q_p\}$
$\Delta = \{\delta_1(section, {q_2}^* {q_p}^*) = q_1,\ \delta_2(section, {q_p}^*) = q_2,\ \delta_3(paragraph, \#PCDATA) = q_p\}$
$F = \{q_1\}$.
L'association d'un état aux nœuds étiquetés *paragraph* est faite de façon directe, la règle de transition δ_3 est appliquée. Toutefois, l'association d'un état aux nœuds étiquetés *section* peut

être faite par deux règles de transition, δ_1 et δ_2. Le choix n'est pas évident car $L(q_2{}^* q_p{}^*) \cap L(q_p{}^*) \neq \emptyset$. Pour la racine de t il est facile de voir que seulement la règle δ_1 est utilisée car le mot $q_2\ q_2\ q_p \notin L(q_p{}^*)$ (règle δ_2). Cependant, pour les nœuds fils de la racine étiquetés *section* les deux règles peuvent être utilisées. □

L'implantation de l'automate d'arbre de l'exemple 4.2.1 peut utiliser l'une des approches suivantes :
(*i*) Associer au nœud visité tous les états possibles. Ensuite, la vérification des pères de ces nœuds est faite : pour chaque règle de transition des nœuds pères, l'automate d'états finis vérifie la concaténation des symboles des nœuds fils, ceci en essayant toutes les combinaisons possibles. Les pères sont associés à leur tour à tous les états possibles. Soit Q_r l'ensemble d'états associé à la racine de l'arbre du document, si $Q_r \cap F \neq \emptyset$ alors l'arbre est accepté par l'automate [MLMK04].
(*ii*) Faire comme dans l'approche ci-dessus mais le résultat de la concaténation des états des nœuds fils est transformé en un automate d'états finis M' et ensuite, l'opération d'intersection est appliquée sur les automates des règles de transitions du nœud père à vérifier et l'automate M'. Le nœud père est associé à tous les états qui correspondent aux règles de transition pour lesquelles l'intersection n'est pas vide.
(*iii*) Associer l'un des états possibles au nœud visité et continuer le processus. Si l'automate échoue dans un nœud ascendant, l'automate revient au nœud où plusieurs états sont possibles et associe un autre état qui n'a pas encore été utilisé (*back tracking*). Le processus reprend jusqu'à ce que l'automate arrive à la racine ou qu'il ne puisse plus associer un état à un nœud [MLMK04].
(*iv*) Transformer l'automate d'arbre non-déterministe en un automate d'arbre déterministe [Chi00].

Comme illustration du problème, nous montrons un exemple de la première approche.

Exemple 4.2.2 Dans la figure 4.4, n est le nœud qui représente la racine de t. On utilise l'automate de l'exemple 4.2.1. Comme le montre l'arbre t' (figure 4.4), les états associés aux nœuds fils de n produisent la séquence $w = \{q_1, q_2\}\ \{q_1, q_2\}\ q_p$. Les règles δ_1 et δ_2 sont utilisées pour associer un état à n car n est étiqueté *section*. L'automate d'états finis M_2 construit à partir de l'expression régulière de δ_2 commence à vérifier w :
(*i*) d'abord M_2 teste si à partir de l'état initial une transition étiquetée q_1 existe, le test échoue.
(*ii*) Ensuite, M_2 fait le même test pour le symbole q_2, le test échoue aussi.
(*iii*) Il n'existe plus de symboles à tester comme premier symbole du mot, alors l'état q_2 ne peut pas être associé au nœud n.
L'automate M_1 de la règle δ_1 n'accepte pas q_1 comme le premier symbole mais accepte q_2, ensuite q_2 est aussi accepté comme deuxième symbole et enfin, q_p est accepté et M_1 est dans son état final, alors l'état q_1 est associé au nœud n. Q_n est l'ensemble d'états associé à n (*i.e.*, $Q_n = \{q_1\}$); comme Q_n n'est pas vide la validation continue. Comme n est la racine, il faut tester si $Q_n \cap F \neq \emptyset$. L'arbre t est accepté par l'automate M_2 car $\{q_1\} \cap \{q_1\} \neq \emptyset$. □

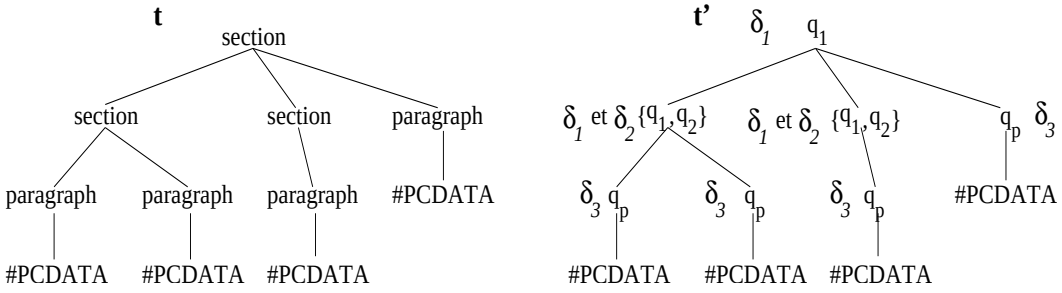


FIG. 4.4 – L'exécution de l'automate M_2 sur l'arbre XML t.

Nous pouvons remarquer que le nombre d'états possibles associés à un nœud n' peut rendre la vérification de la validité du nœud père de n' plus laborieuse car l'automate d'états finis vérifie toutes les règles de transition qui peuvent être utilisées pour chaque état associé à n'.

L'approche proposée dans [HM02, MH03] est fondée sur les automates d'arbres d'arité non-bornée ascendant en prenant compte les attributs. Les auteurs considèrent un schéma équivalent à une grammaire d'arbre régulière. Le langage de schéma choisi est le langage RELAX-NG. Comme le montre la section 3.3, RELAX-NG a une manière très particulière de définir les attributs d'un élément : ils peuvent être représentés par des expressions régulières composées qui mélangent les définitions d'attributs d'un élément e avec les définitions de sous-éléments de e.

La figure 4.5 montre un document XML dont le modèle de contenu de l'élément *article* peut être modélisé par l'expression régulière $author^+\ title\ publisher?$. En prenant en considération les attributs, l'expression régulière composée peut être $@key\ @year?\ author^+\ title\ publisher?$ où les symboles préfixés par le symbole @ représentent les attributs. Comme dans RELAX-NG il est permis de mélanger les attributs et les éléments dans la définition d'un modèle de contenu d'un élément, on pourrait avoir l'expression régulière $(@key|key)\ @year?\ author^+\ title\ publisher?$ qui définit que la valeur *key* peut être définie par un attribut ou par un élément. De plus, comme les expressions régulières sont utilisées pour définir les contraintes structurelles des attributs, on peut avoir une expression comme $@key\ (@year\ @month?|@date)\ author^+\ title\ publisher?$. Dans ce cas, la date de publication de l'article peut être modélisée par deux attributs différents (*year* et *month*) ou par un seul (*date*). Toutes les expressions régulières suggérées décrivent l'élément article du document XML de la figure 4.5.

```
<article key="A01" year="2005">
  <author> ... </author>
  <author> ... </author>
  <title> ... </title>
  <publisher> ... </publisher>
</article>
```

FIG. 4.5 – Un document XML décrivant l'élément *article*.

Le défi dans leur approche est de proposer un automate d'états finis qui traite les attributs et les éléments à la fois. Les validations de modèles de contenus des éléments sont faites par des automates d'éléments[2] [MLMK04] étendus. Un automate d'élément est un automate d'états finis $(Q, \Gamma, \Delta, q_0, Q_F)$ dont l'alphabet est un ensemble de symboles non terminaux d'une grammaire d'arbre régulière G.

L'automate d'élément ne peut pas être utilisé pour valider une expression régulière composée car il faut prendre en compte que la relation d'ordre n'existe pas pour les attributs. Un ensemble d'attributs peut être représenté par une expression régulière standard cependant le nombre de combinaisons à faire est de l'ordre de 2^n (n étant le nombre d'attributs dans l'expression) et l'implantation peut être prohibitive.

Soit une expression régulière composée c définie comme suit :

$$\begin{array}{rcl} c & ::= & @att \\ & & c|c \\ & & c\ c \\ & & c? \\ & & e \end{array}$$

[2]Element automaton.

Où *att* est un nom valide d'attribut d'un schéma et e est une expression régulière sur les éléments.

Étant donné une expression régulière composée E_c, l'automate d'élément-attribut est construit en trois étapes :

1. Un automate d'élément-attribut M_{ea} est construit à partir de E_c (en utilisant un algorithme standard de transformation d'une expression régulière en un automate). Remarquer qu'à ce moment-ci les attributs sont traités comme des éléments.
2. Une déclaration de non-existence d'attributs pour $att(E_c)$ est définie pour être ajoutée à l'automate M_{ea}, où $att(E_c)$ représente l'ensemble d'attributs dans E_c.
3. À partir de l'ensemble d'attributs de l'élément qui doit être vérifié, un automate d'élément M_e est construit à partir de M_{ea} à la volée et seulement la partie du mot composée d'éléments est vérifiée.

Les étapes 1 et 2, ci-dessus, sont exécutées de façon statique, *i.e.*, avant la validation. La troisième étape est exécutée lors de la validation du document. En effet, pour construire l'automate d'élément à partir de l'automate d'élément-attribut d'origine il est nécessaire de connaître les attributs définis pour l'élément qui est en train d'être testé.

Il est important de remarquer que chaque élément d'un document à valider donne origine à un mot construit par la concaténation des ses attributs et des ses éléments. Un automate M_e est construit pour chacun de ces mots.

Nous présentons, d'abord, les étapes 1 et 3 car le résultat est utilisé pour expliquer l'étape 2 de façon plus intuitive. L'étape 1 construit l'automate à partir de l'expression régulière composée en utilisant un algorithme de transformation standard [HMU01]. Par exemple, étant donné l'expression régulière composée $E_c = (@key|key)\ @year?\ author^{+} title\ publisher?$ qui peut modéliser l'élément *article* (figure 4.5), l'automate d'élément M_{ea} résultant est celui présenté dans la figure 4.6.

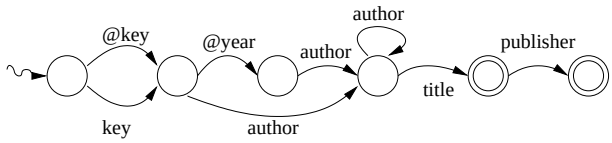


FIG. 4.6 – L'automate d'élément-attribut construit à partir de $(@key|key)\ @year?\ author^{+}\ title\ publiher?$ sans utiliser l'étape 2.

L'étape 3 est exécutée pendant la validation du document comme suit : Soient t un arbre XML et n un nœud étiqueté e qui représente un élément dans t. Soit α l'ensemble d'étiquettes des nœuds du type attribut, fils de n. Les transitions de l'automate M_{ea} ayant les étiquettes $@a$ telles que $a \in \alpha$ sont remplacées par des transitions étiquetés ε (vides) dans M_e et les autres transitions $@a'$ sont supprimées de M_e. L'automate résultant de l'étape 3 (l'automate M_e) n'a plus d'arcs étiquetés avec des étiquettes qui représentent les attributs : soit les étiquettes sont remplacées par ε soit les arcs sont supprimés. Par exemple, si les fils d'un nœud étiqueté *article* composent le mot $w = @key\ @year\ author\ title$ (c'est-à-dire, $\alpha = \{key, year\}$) l'automate M_{ea} de la figure 4.6 est modifié donnant lieu à l'automate M_e (figure 4.7).

Le mot à vérifier par l'automate M_e n'est pas le mot d'origine car l'automate M_e vérifie seulement les étiquettes qui correspondent aux éléments. Ainsi, l'automate M_e de la figure 4.7 vérifie seulement la partie de w qui correspond aux éléments, c'est-à-dire, $w_e = author\ title$ qui est accepté par M_e. Toutefois, supposons maintenant le mot $w' = @key\ @year\ key\ author\ title$. L'automate M_e construit à partir de w' et M_{ea} accepte w' qui n'est pas décrit par l'expression régulière E_c. Remarquer que M_e vérifie la partie $w'_e = key\ author\ title$. Le mot w' a les symboles

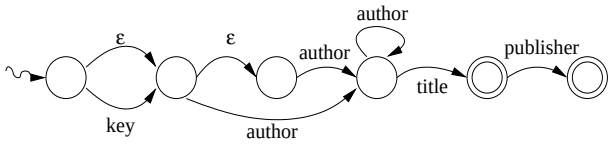


FIG. 4.7 – L'automate résultant de l'application du pas 3 sur l'automate de la figure 4.6.

@*key* et *key* et les mots décrits par E_c doivent avoir soit le symbole @*key*, soit le symbole *key* et non les deux. Ensuite considérons le mot $w'' =$ @*foo key author title* et sont M_e correspondant. Le mot w'' a un autre problème, l'attribut @*foo* n'est pas défini par l'expression régulière E_c et pourtant M_e accepte la partie de $w''_e =$ *key author title* composée par des éléments. Rappelons que si les transitions étiquetées @*key* et @*year* sont supprimées de M_{ea} (figure 4.6), il existera, à partir de l'état initial de M_e, un arc étiqueté *key*.

Pour résoudre ce problème, il faut mettre en place l'étape 2. Cette étape construit la déclaration de la non-existence d'attributs, notée par !@N. Cela signifie qu'aucun attribut dans l'expression régulière composée (utilisée pour construire l'automate d'élément-attribut de l'étape 1) n'a un nom dans l'ensemble N. Cette déclaration garantit la non présence des attributs de !@N dans les mots qui appartiennent au langage décrit par une expression régulière composée E_c.

Par exemple, reprenons l'expression $E_c =$ (@*key*|*key*) @*year*? *author*$^+$ *title publisher*?, si la déclaration de non-existence est introduite, E_c devient $E'_c =$!@($N \setminus \{key, year\}$) (@*key* | !@*key key*) @*year*? *author*$^+$ *title publisher*?. Remarquer que $N \setminus \{key, year\}$ définit tous les attributs acceptables sauf *key* et *year*. Le symbole !@*key* définit que s'il existe un attribut *key* la transition représentée par !@*key* est supprimée dans l'automate construit à partir de E'_c, sinon l'étiquette de la transition est remplacée par ε. La figure 4.8 présente l'automate résultant de l'insertion de la non-existence d'attributs. En utilisant l'étape 2, l'automate de la figure 4.8 remplace celui de la figure 4.6.

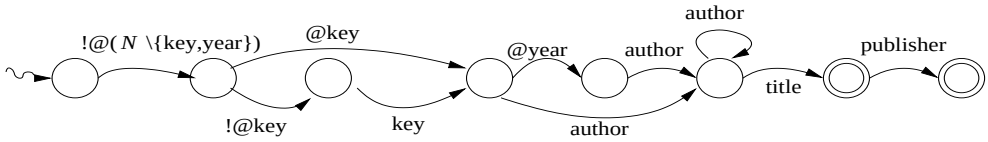


FIG. 4.8 – L'automate d'élément-attribut construit à partir de (@*key*|*key*) @*year*? *author*$^+$ *title publisher*? en utilisant l'étape 2.

Si on reprend le mot $w =$ @*key* @*year author title*, la transformation de l'automate d'élément-attribut M_{ea} de la figure 4.8 en l'automate d'élément M_e est faite comme suit : (*i*) l'étiquette de la transition qui sort de l'état initial est remplacée par vide (ε) car $\alpha \not\subset N \setminus \{key, year\}$ (α étant l'ensemble d'attribut qui apparaît dans w) (*ii*) comme il existe un attribut étiqueté *key* la transition étiquetée !@*key* de M_{ea} est supprimée. Le mot w_e est encore accepté par M_e. Considérons maintenant le mot $w'' =$ @*foo key author title* et son automate M_e. Le mot w' n'est plus accepté car la transition étiquetée !@($N \setminus \{key, year\}$) a été supprimée de l'automate M_{ea}, donnant l'automate M_e et, par conséquent, aucun mot n'est accepté par M_e. Le mot $w' =$ @*key* @*year key author title* n'est pas non plus accepté car la transition étiquetée !@*key* est supprimée et ainsi, la transition étiquetée *key* ne pourra jamais être exécutée.

Dans [HM02] la preuve de la correction de l'algorithme est présentée mais son implantation reste très laborieuse car il faut créer l'automate en trois étapes. De plus, pour chaque groupe d'attributs il faut construire l'ensemble de non-existence d'attributs. La complexité de la mé-

thode présentée n'a pas été étudiée mais les auteurs affirment que la performance est raisonnable dans les exemples qu'ils utilisent.

Une autre façon de valider les documents XML via des automates d'arbre est d'utiliser l'API DOM [WHA+00] (*Document Object Model*). Cette API est fondée sur un modèle d'arbre : le document XML est d'abord chargé dans la mémoire vive et ensuite, il peut être parcouru en utilisant des algorithmes de parcours d'arbres. L'une des implantations de *DOM* en Java exécute aussi la validation d'un document XML. Cependant il n'existe pas de documentation permettant de l'analyser.

4.3 La validation incrémentale des documents XML

Lorsqu'un document XML X, valide par rapport à un schéma d, est mis à jour, il faut vérifier si le nouveau document X' reste valide. Il serait intéressant de ne pas valider X' en entier car une bonne partie de X n'aura pas changé.

La validation incrémentale peut être décrite comme suit : Étant donné un schéma d, un arbre XML t et une séquence de mises à jour qui transforme t en un autre arbre XML t', vérifier si t' est encore valide par rapport à d d'une façon telle que le coût de vérification soit inférieur à celui de la validation du document entier [PV03]. Généralement, la validation incrémentale s'appuie sur les informations connues des validations précédentes pour recalculer uniquement ce qui a changé.

Nous présentons les primitives de mise à jour utilisées dans [BH03] qui nous serviront pour présenter trois méthodes de validation incrémentale. Soient un arbre XML t et une opération de mise à jour exécutée sur une position p donnée de t.

- $insert(t_p, p, t)$: Exécute l'insertion dans t d'un sous-arbre non vide, noté t_p, à la position $p \in fr^{ins}(t)$.
- $insertBefore(t_p, p, t)$: Exécute l'insertion dans t d'un sous-arbre non vide, noté t_p, à la position $p \in dom(t)$. Dans ce cas, tous les sous-arbres de t ayant leur racine à la position p ou à une position *frère droit* de p, sont poussés à droite, *i.e.*, leurs positions sont incrémentées de 1. La racine du sous-arbre t_p est placée à la position p.
- $delete(p, t)$: Exécute la suppression d'un sous-arbre non vide de l'arbre t enraciné à la position p et noté t_p. Tous les sous-arbres de t ayant comme position racine un frère droit de p sont tirés à gauche, *i.e.*, leurs positions sont décrémentées de 1.
- $replace(t_p, p, t)$: Exécute le remplacement dans t d'un sous-arbre non vide enraciné à la position p par un nouveau sous-arbre non vide t_p.

La figure 4.9 illustre chacune de ces opérations.

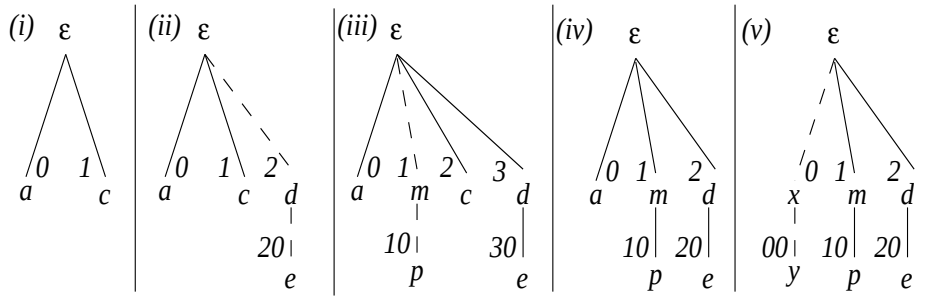


FIG. 4.9 – (*i*) L'arbre XML d'origine (*ii*) Une insertion dans la position $p = 2$ (*iii*) Une insertion avant $p = 1$ (*iv*) Suppression dans $p = 2$ (*v*) Remplacement dans $p = 0$

Étant donné une opération de mise à jour $ope(t_p, p, t)$ et un nœud n tel que le nœud dans la position p est fils de n, la validation incrémentale est faite comme suit :
(i) si l'automate d'arbre est déterministe, il faut vérifier seulement si la règle de transition appliquée au nœud n associe encore un état valide à n, ou
(ii) si l'automate d'arbre est non-déterministe, il faut vérifier les états de tous les nœuds dans le chemin de la racine jusqu'à n. Si l'état associé à la racine, après la vérification, appartient encore à l'ensemble d'états finaux, la mise à jour est valide.

Les approches de validation incrémentale s'appuient en général sur des structures auxiliaires pour proposer des méthodes efficaces [BH03, PV03, BPV04]. Ces structures ont des informations sur les arbres XML avant les mises à jour et ainsi, ces informations sont confrontées avec les informations sur les mises à jour.

Dans [PV03], il est proposé une méthode de validation incrémentale qui prend en considération des schémas sans attributs modélisés par des grammaires d'arbres régulières. Dans leur approche, un document XML X a un arbre binaire t_b associé. Cet arbre binaire est construit selon les pas montrés dans la section 4.2. Remarquer que comme ils utilisent les grammaires d'arbres régulières, l'application μ est définie pour faire correspondre les types (symboles non-terminaux) aux étiquettes des nœuds (symboles terminaux).

Exemple 4.3.1 Étant donné l'arbre t de la figure 4.10 qui représente un document XML (où *data* représente un nœud avec des données), l'arbre t_b de la figure 4.10 est la représentation binaire de t. L'automate d'arbre $M = (\Sigma, Q, \Delta, F, \mu)$ accepte t, où :
(i) $\Sigma = \{garagiste, occasion, neuve, annonce, annee, modele\}$
(ii) $Q = \{garagiste, occasion, neuve, annonceOcc, annonceNeu, annee, modele\}$
(iii) $\Delta = \{(garagiste, occasion\ neuve) \rightarrow garagiste, (occasion, annonceOcc^*) \rightarrow ocassion,$
$(neuve, annonceNeu^*) \rightarrow neuve, (annonce, annee\ modele) \rightarrow annonceOcc,$
$(annonce, annee?\ modele) \rightarrow annonceNeu, (annee, data) \rightarrow annee,$
$(modele, data) \rightarrow modele\}$
(iv) $F = \{garagiste\}$
(v) $\mu(garagiste) \rightarrow garagiste, \mu(occasion) \rightarrow occasion, \mu(neuve) \rightarrow neuve,$
$\mu(annonceOcc) \rightarrow annonce, \mu(annonceNeu) \rightarrow annonce, \mu(annee) \rightarrow annee,$
$\mu(modele) \rightarrow modele.$
Les nœuds dans les positions 00, 10, 11 et 13 de t sont associés aux états $annonceOcc$, $annonceNeu$, $annonceNeu$ et $annonceNeu$, respectivement. □

L'approche de [PV03] est basée sur un arbre binaire t_b pour vérifier, de façon incrémentale, si les mises à jour sur un document XML sont valides. En utilisant l'arbre t_b, les auteurs proposent une deuxième structure auxiliaire qui pointe vers des chemin précis dans t_b. Ces chemin sont appelés lignes, notés $line(n)$ (où n est un nœud de t_b), et sont définis comme suit :

- D'abord une ligne $line(\varepsilon)$, appelée *ligne principale*, est construite à partir de la racine de t_b :
  (i) le nœud racine appartient à $line(\varepsilon)$.
  (ii) les nœuds internes qui appartiennent à $line(\varepsilon)$ sont calculés de la façon suivante :
  Soient v_1 et v_2 les nœuds fils de la racine de t_b.
  Si $|v_1| \geq |v_2|$ (où $|v|$ signifie de nombre de nœuds descendants d'un nœud v), alors v_1 participe de $line(\varepsilon)$, sinon v_2 participe de $line(\varepsilon)$. C'est-à-dire, le nœud fils qui a le plus grand sous-arbre devient le prochain membre de $line(\varepsilon)$.
  Pour les nœuds fils de v_1 (ou v_2) la même stratégie est appliquée. Le processus s'arrête lorsqu'un nœud feuille est trouvé.

- Après le calcul de la ligne principale $line(\varepsilon)$, chaque nœud membre de $line(\varepsilon)$ sera la racine pour le calcul d'autres lignes, appelées *lignes secondaires*, de la même façon que pour le

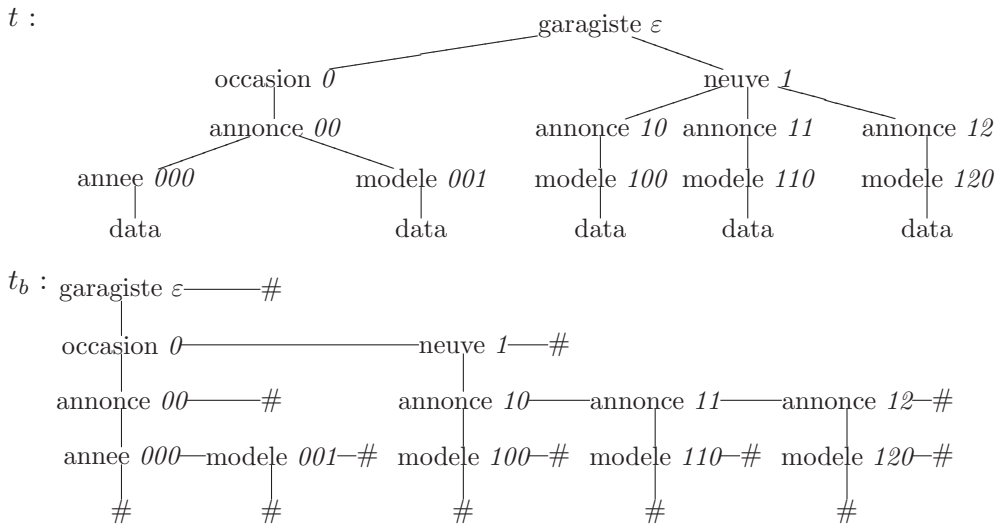


FIG. 4.10 – L'arbre XML (avec des positions) qui stocke des donnés sur les annonces de voitures et sa version binaire.

calcul de $line(\varepsilon)$. Les nœuds qui participent de $line(\varepsilon)$ étiquetés # ne sont pas utilisés dans les calculs des lignes secondaires.

Exemple 4.3.2 Étant donné l'arbre binaire t_b de la figure 4.10, la ligne principale est
$line(garagiste) = garagiste/occasion/neuve/annonce/annonce/annonce/modele/\#$
(par positions $\varepsilon/0/1/10/11/12/120/\#$) et si on prend le nœud *occasion* de la $line(garagiste)$, le résultat est $line(occasion) = occassion/annonce/annee/modele/\#$. □

Pour chaque ligne calculée l, un automate d'états finis M_l qui reconnaît les mots qui peuvent apparaître dans l est construit. Le résultat de l'exécution de M_l sur l est stocké dans une structure auxiliaire sous la forme d'un arbre, appelé arbre de transition, notée T_l, qui est construit en utilisant une variation de la méthode *B-tree* [CLR00]. Cette variation de *B-tree* construit un arbre 2-3, *i.e.*, un arbre dont chaque nœud contient 3 cellules. Chaque cellule c de T_l contient soit la plage d'états du mot représenté par l accessible à partir de c soit le vide (chaque nœud peut avoir, au maximum, une cellule vide si $\mid w \mid \geq 2$). Les cellules non-vide peuvent être dans un nœud feuille ou avoir un nœud fils. Le but de construire l'arbre de transition est de vérifier, après une mise à jour d'un arbre XML, seulement les états concernés par la position mise à jour. Lorsqu'une mise à jour est exécutée, l'arbre t_b associé à l'arbre XML mis à jour est aussi mis à jour. Après la mise à jour de t_b, l'arbre T_l associé à la ligne l de t_b affectée par la mise à jour est aussi mis à jour.

La figure 4.11 donne l'intuition de comment l'arbre de transition est construit à partir d'un mot qui représente $line(occasion)$ de l'exemple 4.3.2 (qi étant l'état initial de l'automate qui reconnaît $line(occasion)$).

La mise à jour de T_l après une insertion est faite comme suit :
- Une cellule d'un nœud feuille de T_l doit pointer vers le symbole inséré. L'arbre T_l est mis à jour pour avoir cette nouvelle cellule.
- La cellule qui pointe vers le nouveau symbole stocke l'état de ce symbole. Les nœuds ascendants du nœud où la cellule a été crée sont mis à jour aussi.
- Si les cellules dans la racine de T_l contiennent l'état initial et l'un des états finaux de l'automate, alors la mise à jour de l'arbre XML est valide.

Pour des suppressions, les principes sont les mêmes.

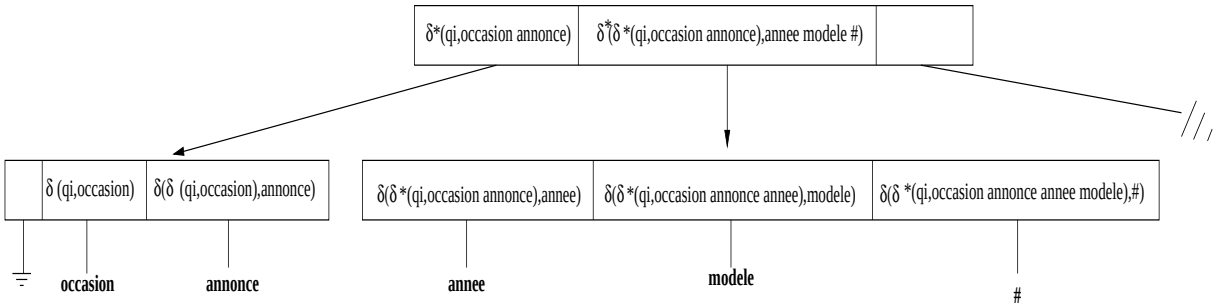
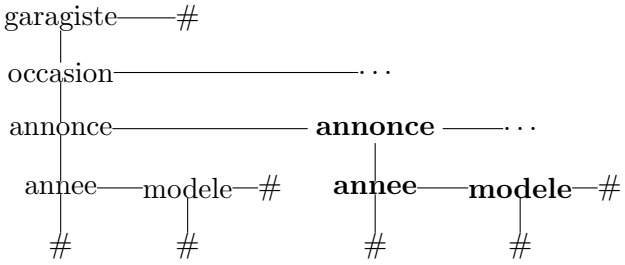


FIG. 4.11 – L'arbre de transition pour $line(occasion)$.

Exemple 4.3.3 Soit t_b l'arbre binaire de la figure 4.10. Soit $E = occasion\ (annonce\ annee\ modele)^*$ l'expression régulière qui décrit $line(occasion)$. Soit M_{occ} l'automate d'états finis construit à partir de E. Soit $w = q_{occasion}\ q_{annonce}\ q_{annee}\ q_{modele}\ q_{\#}$ le résultat de l'exécution de l'automate M_{occ} sur $line(occasion)$. Soit T_{occ} l'arbre de relation de $line(occasion)$ (figure 4.11). Étant donné l'opération de mise à jour $insert(01, t, t_{annonce})$ (où $t_{annonce}$ est sous-arbre ayant la racine étiquetée *annonce*) sur l'arbre t de la figure 4.10, l'arbre binaire t_b (figure 4.10) est mis à jour pour produire le nouveau sous-arbre $t_b{}'$ dont le morceau concerné par la mise à jour est présenté ci-dessous :



La ligne secondaire $line(occasion)$ contient maintenant le chemin suivant

$$occasion/annonce/\texttt{annonce}/annee/modele/\#.$$

L'arbre T_{occ} est aussi mis à jour en utilisant M_{occ}, produisant T'_{occ} : une cellule avec δ^* (δ^* ($q_i, occasion\ annonce$), $annonce$) dans un nœud feuille est crée dans T'_{occ} à gauche de la cellule qui contient δ^* (δ^* ($q_i, occasion$), $annonce$). La vérification de la validation est faite en vérifiant si la composition des cellules de racine de T'_{occ} contient une paire (q_0, q_f). L'automate d'arbre M (de l'exemple 4.3.1) associe un nouveau type au nœud étiqueté *occasion* dans la position 0. Si le type du nœud dans la position 0 ne change pas alors la mise à jour est valide. Si le type change pour un autre type valide, alors il faut exécuter les mêmes pas pour la ligne principale et ainsi, il faut vérifier si l'état associé à la racine est un état final. Si le type associé au nœud dans la position 0 est $\perp$, alors la mise à jour est invalide. Comme l'état de la racine de $line(occasion)$ ne change pas, la ligne principale à partir du nœud étiqueté *occasion* n'est pas recalculée. Ainsi, l'état de la racine ne change pas et donc, la mise à jour est valide. □

La méthode proposée dans [PV03] a des limites :

1. Il n'existe pas une manière efficace de traiter des mises à jour multiples : pour une séquence de m mises à jour, la modification des structures auxiliaires est faite pour chaque mise à jour et la vérification est exécutée après la m-ème mise à jour.
2. Parfois il est plus efficace de valider le document entier qu'utiliser la méthode proposée car elle devient très coûteuse.
3. Les mises à jour traitées sont toujours sur un nœud simple, *i.e.*, insertion et suppressions de nœuds feuilles et renommage.

Dans [BPV04], les auteurs présentent une version étendue de l'article ici présenté. Dans la version étendue ils proposent cette méthode pour plusieurs formalismes de schéma : DTD, XML-Schema et RELAX NG. Les algorithmes sont les mêmes que ceux présentés ci-dessus, complétés par une étude de performance.

Les méthodes proposées dans [BH03, BML$^+$04] sont similaires : elles sont basées sur le formalisme de schéma correspondant à la grammaire d'arbre régulière locale (*i.e.*, des DTD) et considèrent les contraintes des valeurs imposées par les attributs des types ID et $IDREF/IDREFS$. L'approche de [BH03] traite en plus les mises à jour des attributs.

Dans [BH03], un automate d'arbre d'arité non-bornée étendu pour le traitement des attributs [BDHL03] est utilisé pour valider les documents XML. Cet automate a la forme $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, où Q est l'ensemble d'états, Σ est l'alphabet[3], Q_f est l'ensemble d'états finals ($Q_f \subseteq Q$) et Δ est l'ensemble de règles de transitions de la forme $a, S, E \rightarrow q$ où
(*i*) $a \in \Sigma$ représente l'étiquette des nœuds de l'arbre t,
(*ii*) $S = \langle S_{compulsory}, S_{optional} \rangle$ (avec $S_{compulsory} \cup S_{optional} \subseteq Q$) où $S_{compulsory}$ représente les attributs obligatoires et $S_{optional}$ les attributs optionnels,
(*iii*) E est une expression régulière sur Q et représente la contrainte de sous-éléments de l'élément a, et
(*iv*) $q_a \in Q$ représente l'état à associer qu nœud de t étiqueté a.

Dans [BH03], deux tables V_{ID} et V_{IDREF} stockent les valeurs des attributs du type ID et $IDREF/IDREFS$, respectivement. Un arbre XML est valide si l'état associé à la racine appartient à Q_f, la table V_{ID} n'a que des valeurs uniques et toutes les valeurs de la table V_{IDREF} ont une valeur correspondante dans la table V_{ID}.

La validation est faite en exécutant une règle de transition δ sur le nœud père n de la position mise à jour. Pour cela, un automate d'états finis qui accepte le langage de l'expression régulière dans δ est exécuté sur le mot construit à partir de la concaténation des états associés aux fils élément de n du type élément (si la racine du sous-arbre inséré est du type élément). Si la racine du sous-arbre est du type attribut, alors un test d'appartenance est exécuté en utilisant l'ensemble S de δ.

La méthode proposée dans [BML$^+$04] améliore la méthode décrite dans [BH03] : l'automate d'états finis n'est pas exécuté sur le mot entier mais, uniquement, sur la position du mot mis à jour. Cette position est calculée à partir de la position de l'opération de mise à jour. Pour cela, une structure auxiliaire qui stocke l'état de chaque symbole dans la position i d'un mot w ($1 \leq i \leq m$) qui représente la concaténation des fils d'un nœud dans un arbre XML, *i.e.*, $\hat{\delta}(q_0, w[1:i])$ est utilisée. Pour vérifier, par exemple, si une insertion d'un symbole α à la fin du mot w est valide, il suffit de tester si $\delta(\hat{\delta}(q_0, w), \alpha) \in F$. De même pour une position au milieu du mot d'origine : si un nouveau symbole α est inséré dans un mot $w[1:n]$ dans la position i ($1 \leq i < n$) il suffit de tester si $\delta(\hat{\delta}(q_0, w[1:i-1]), \alpha) \in Q$ (où Q est l'ensemble des états de l'automate d'états finis utilisé pour accepter w).

Cette méthode a une limite[4], si l'expression régulière a des symboles répétés alors l'utilisation de la structure auxiliaire n'est pas efficace. Par exemple, étant donné l'expression régulière $E = a(b^*|cb^*)$, un mot $w = acbbb$ qui appartient à $L(E)$ et la structure auxiliaire $w_q = q_a q_c q'_b q'_b q'_b$ de w (q_a étant l'état qui représente a, q_c l'état qui représente c et q'_b l'état qui représente le deuxième b de E), si le c est supprimé de w les états stockés dans w_q pour les b après le c supprimé ne sont plus q'_b car, après la mise à jour, ils correspondent au premier b de E et non plus au deuxième, c'est-à-dire que le mot w_q doit stocker les états $q_a q_b q_b q_b$ (q_b étant l'état qui représente le premier b de E). Dans ce contexte, la complexité de la validation revient à la

[3] Σ est composé par l'alphabet selon la définition 3.3.2.
[4] Sans compter le coût de maintenance et la taille de la structure auxiliaire.

complexité de la validation du mot entier.

Pour résoudre cela, les DTD sont classées[5] [BML+04] :
(*i*) Les DTD qui n'ont pas de modèles de contenu avec des symboles répétés, appelée *conflict-free DTD*, et
(*ii*) les DTD qui ont des modèles de contenu avec des symboles répétés cependant les symboles répétés ont toujours entre eux, au minimum, deux autres symboles non-répétés, appelée *1,2-conflict-free DTD*.

La méthode présentée ci-dessus n'est efficace que pour les *conflict-free DTD*, pour les *1,2-conflict-free DTD* une autre méthode est proposée. Deux nouvelles notions sont donc introduites :
(*i*) les mots qui peuvent avoir n symboles supprimés sans que les états existants dans la structure auxiliaire soient modifiés. Ces mots sont appelés *contractible*, et
(*ii*) les mots qui peuvent avoir n symboles insérés sans que les états existants dans la structure auxiliaire soient modifiés. Ces mots sont appelés *expansible*.
Étant donné une expression régulière E non-ambiguë et $w[1:k] \in L(E)$, un mot w est i-contractible s'il est possible de supprimer $w[i]$ de w sans recalculer la structure auxiliaire, *i.e.*, $\hat{\delta}(q_0, w[1:i-1]w[i+1]) = \hat{\delta}(q_0, w[1:i-1]w[i]w[i+1])$. De même, a mot w est a-expansible (où a est un symbole dans l'alphabet de E), s'il est possible d'insérer a dans la position i de w sans recalculer la structure auxiliaire, *i.e.*, $\hat{\delta}(q_0, w[1:i-1]aw[i]) = \hat{\delta}(q_0, w[1:i-1]w[i])$. La validation incrémentale des *1,2-conflict-free DTD* d est réduite à trouver si le mot du langage décrit par une expression régulière E est i-contractible ou a-expansible pour les suppressions ou les insertions, respectivement.

La validation des valeurs du type ID est faite de façon directe :
- Insertions : s'il est inséré des attributs du type ID, leurs valeurs ne peuvent pas exister dans la table de valeurs ID de l'arbre XML à mettre à jour. Si les valeurs ID sont uniques, ils sont insérés dans la table de valeurs ID [BH03, BML+04].
- Suppressions : s'il est supprimé des attributs du type ID, leurs valeurs ne peuvent pas être référencées par des attributs du type $IDREF/IDREFS$ dans l'arbre XML à mettre à jour. Dans [BH03], la table V_{IDREF} est vérifiée : si l'une des valeurs des attributs ID à supprimer existe dans V_{IDREF}, alors la suppression est invalide. Dans [BML+04], la table qui stocke les valeurs ID a un compteur qui stocke le nombre de valeurs $IDREF/IDREFS$ qui font référence à une valeur ID : si le compteur d'un attribut ID est égal à zéro, alors l'attribut peut être supprimé, sinon la suppression est invalide. Si les conditions de suppression sont respectées, les valeurs ID sont supprimées de la table qui stocke les valeurs ID [BH03, BML+04].

La validation des valeurs du type $IDREF/IDREFS$ est la plus simple :
- Insertions : s'il est inséré des attributs du type $IDREF/IDREFS$, leurs valeurs doivent avoir des valeurs correspondantes dans la table qui stocke les valeurs ID. Si la condition est vérifiée, le compteur de nombre de références est augmenté [BML+04] ou les valeurs $IDREF/IDREFS$ sont stockés dans la table V_{IDREF} [BH03].
- Suppressions : les valeurs $IDREF/IDREFS$ sont supprimées de la table V_{IDREF} [BH03] ou le compteur des valeurs ID correspondantes sont décrémentés [BML+04].

Bien que la validation incrémentale soit très importante dans le contexte de mises à jour de documents XML, les travaux présentés dans cette section ont des approches qui traitent une mise à jour à la fois. Les trois approches annoncent comme extensions le traitement des mises à jour multiples.

[5] Remarquer que ces classes n'ont aucun rapport avec l'ambiguïté des expressions régulières. En fait, les DTD considérées-ci sont non-ambiguës.

Chapitre 5

Évolution et changement des schémas

Dans ce chapitre, nous nous intéressons aux changements des structures des bases de données XML, c'est-à-dire, nous discutons l'évolution des schémas XML (dus à une adaptation requise par le monde extérieur, *e.g.*, l'évolution d'une application pour son adaptation aux nouvelles règles du marché, etc).

La mise à jour des schémas XML n'est pas une tâche triviale car il faut prendre en considération tous les documents valides par rapport à un schéma avant la mise à jour. En effet, pour que ces documents restent valides par rapport au nouveau schéma, il faut vérifier s'ils ont besoin d'être modifiés [AE03].

Nous pouvons partager les méthodes pour l'évolution des schémas XML en deux grands groupes :

- L'évolution de schémas qui préserve la cohérence de la base de données XML sans modifier les documents préexistants. En d'autres termes, les documents valides par rapport au schéma d'origine ne sont pas changés et restent valides par rapport au nouveau schéma.
- L'évolution de schémas qui modifie les documents valides préexistants pour préserver la cohérence de la base de données.

Bien que l'évolution de schémas ait été identifiée comme un mécanisme souhaitable pour les bases de données XML [RAJB+00, CS00], il n'y a pas, à notre connaissance, un nombre significatif de travaux de recherche dans cet axe. La majorité des approches travaillent sur la transformation des documents XML, dans le cadre de publication, de transformation et d'intégration de données.

Ce chapitre est donc divisé en quatre sections : la première section présente les approches de vérification de typage et d'inférence de types (où types veut dire schémas), la deuxième partie montre les méthodes d'adaptation des documents aux schémas, la troisième section présente les approches qui proposent des primitives de mises à jour des schémas XML et, pour terminer, nous allons présenter dans la quatrième partie le problème d'inférence grammaticale (ou apprentissage d'automate d'états finis) en faisant un rapport avec l'évolution des schémas.

5.1 Vérification de typage et inférence de type

La vérification de typage est similaire à la vérification de la validité d'un document XML [KSS03, MBPS05, Suc02b, Via03]. La vérification de la validité est exécutée comme suit : étant donné un arbre XML t et un schéma (type) d, vérifier si $t \in L(d)$. Dans le cadre de la vérification

de typage, au lieu de vérifier l'appartenance d'un arbre à un langage d'arbre, on vérifie si tous les résultats d'un programme exécuté sur des arbres XML appartiennent à un schéma donné.
Soient (*i*) $\mathcal{X}$ un ensemble de documents XML,
(*ii*) d un schéma, appelé schéma d'origine, tel que $\mathcal{X} \subseteq L(d)$,
(*iii*) d' un schéma cible, et
(*iv*) F un programme exécuté sur $\mathcal{X}$.
Vérifier si : $X \in \mathcal{X} \Rightarrow F(X) \in L(d')$.
Le vérificateur de typage analyse le programme et décide si tous les documents produits sont conformes au schéma cible ou non. On note $F(L(d))$ l'ensemble de documents transformés par F ayant comme entrée les documents valides par rapport à d.

On peut comparer le vérificateur de typage du contexte XML à un vérificateur de type d'un langage de programmation (fortement typé) : lorsqu'un programme est compilé, le compilateur doit vérifier si les règles pour chaque type utilisé sont respectées. Par exemple, une variable du type entier doit être affectée d'une valeur entière. Si toutes les règles sont respectées, alors le programme est accepté.

Les programmes dans lesquels la vérification de typage est appliquée sont classés comme suit [Suc02b] :

- Programmes qui transforment les données dans une base de données relationnelle en documents XML, appelés *programmes de publication*.
- Programmes qui transforment un document XML en d'autre documents XML, appelés *programmes de transformation*.
- Programmes qui transforment deux ou plusieurs documents XML en un autre document XML, appelés *programmes d'intégration*.

La majorité des programmes sont de la classe des programmes de transformation. Les formalismes les plus utilisés pour transformer les documents XML sont [Suc02b] : XSLT (eXtensible Style Language Transformation [Cla99]) et XQuery (XML Query [CFR+01]).

L'une des approches le plus utilisée pour résoudre le problème de vérification de typage est l'inférence de type [KSS03, PV00, Suc02b, Via03]. L'inférence de type est énoncé comme suit : étant donné un schéma d et un programme F, un programme d'inférence de type $\overline{F}$ infère un schéma d'' à partir de d et des résultats du programme F tel que $L(d'') \subseteq F(L(d))$.
Si $F(L(d'')) \subseteq L(d)$, alors $\overline{F}$ est correct; si $F(L(d'')) = L(d)$, alors $\overline{F}$ est correct et complet.
Si $\overline{F}$ est complet et correct, l'inférence de type peut être utilisée pour résoudre le problème de vérification de typage : en effet, pour vérifier si $F(L(d)) \subseteq L(d')$, il suffit de vérifier si $L(d'') \subseteq L(d')$ (où d' est le schéma cible).

Exemple 5.1.1 Étant donné le document XML X de la figure 2.1 et la DTD D de la figure 3.2, supposons un programme F qui transforme X en un document X' qui contient seulement les articles publiés en 2000 (leurs sujets et leurs titres) de X. Le résultat de l'exécution de F sur X a la structure suivante :

```
<Resultat>
  <Pub2000>
     <Sujet> ...   </Sujet>
     <TArticle> ... </TArticle>
  </Pub2000>
  <Pub2000>
     <Sujet> ...   </Sujet>
     <TArticle> ... </TArticle>
  </Pub2000>
       :
</Resultat>
```

Supposons que l'application d'un programme d'inférence $\overline{F}$ à partir de F et D infère la DTD D' :

```
<!ELEMENT Resultat (Pub2000*)>
<!ELEMENT Pub2000  (Sujet,TArticle)>
<!ELEMENT Sujet    (#PCDATA)>
<!ELEMENT TArticle (#PCDATA)>
```

Dans ce cas $F(L(D)) = L(D')$. Étant donné un schéma D_{out}, pour le problème de vérification de typage il suffit de vérifier si $L(D') \subseteq L(D_{out})$. □

L'inférence de type ne permet pas de résoudre tous les cas de problème de vérification de typage car les résultats des programmes F peuvent ne pas être modélisés par des expressions régulières.

Par exemple, soit d un schéma d'origine modélisé par $root \rightarrow \mathbf{root}(p^*)$. Les documents X conformes à d ont la forme :

```
<root> <p>...</p>...<p>...</p></root>
```

Soit F le programme suivant (en XQuery) :

```
<result>
  {
    FOR $x IN $root RETURN <a> data{$x/p} </a>,
    FOR $x IN $root RETURN <b> data{$x/p} </b>
  }
</result>
```

L'exécution de F sur X donne :

```
<result> <a>...</a><b>...</b> ... <a>...</a><b>...</b> </result>
```

Dans ce contexte, les éléments a et b doivent se répéter le même nombre de fois dans les documents résultats de l'exécution de F. Le programme d'inférence infère le schéma d'' : $result \rightarrow \mathbf{result}(a^*b^*)$ car a^nb^n n'est pas une expression régulière ($n \geq 0$). Si le schéma cible d' est $result \rightarrow \mathbf{result}(aa^*bb^*)$, alors $L(d') \not\subseteq L(d'')$ et le programme de vérification de typage par rapport à d' rejette F malgré le fait que F soit correct. Ainsi, un système de vérification de typage qui utilise l'inférence de type peut rejeter des programmes qui sont corrects.

Une option pour utiliser l'inférence de type dans le cadre de vérification de typage est d'utiliser l'inférence *inverse* de type [MSV00], c'est-à-dire, en utilisant le schéma cible, on exécute un programme d'inférence inverse de type $\overline{F}^{-1}$. L'inférence inverse de type est énoncée comme suit : étant donné un programme F, un schéma d'origine d et schéma cible d', d'abord $\overline{F}^{-1}(d')$ est calculé et ensuite, on vérifie si $L(d) \subseteq L(\overline{F}^{-1}(d'))$.

Si on reprend l'exemple ci-dessus, en considérant que d' : $result \rightarrow \mathbf{result}(aa^*bb^*)$ est le schéma cible, alors l'inférence inverse de type $\overline{F}^{-1}$ infère, à partir de d' et F, le schéma d'^{-1} : $root \rightarrow \mathbf{root}(pp^*)$. Comme $L(d) \subseteq L(d'^{-1})$, alors le programme F est accepté.

Bien que l'utilisation de l'inférence (inverse) de type augmente les classes de programmes de transformation dont la vérification de typage est possible, il existe encore des classes de programmes dans lesquelles la vérification n'est pas possible [KSS03, Via03] :

– La vérification ne traite pas des valeurs dans les programmes.

– Les programmes ne peuvent pas avoir de jointures.
– Le vérificateur n'arrive pas à montrer où le programme de transformation échoue par rapport au schéma cible.

Parmi les formalismes qui s'appuient sur l'inférence de type pour faire la vérification de type pour la transformation de documents, on peut citer XQuery, XDuce [HP03], ℂDuce [BCF03]. XDuce et ℂDuce sont de la famille des langages fonctionnels fortement typés destinés à manipuler les documents XML.

La vérification de typage peut être utilisée pour l'évolution de schéma de la façon suivante : étant donné un ensemble de documents XML $\mathcal{X}$ et un schéma d tel que $\mathcal{X} \subseteq L(d)$. Supposons que le schéma d soit mis à jour, donnant le schéma d'. Il faut construire un programme F tel que $\forall X \in \mathcal{X}$, $F(X) \in L(d')$. Le programme F fait la mise à jour de l'ensemble $\mathcal{X}$ pour que tous les documents soient valides par rapport au nouveau schéma d'.

L'inconvénient d'utiliser cette approche pour l'évolution de schéma en préservant la cohérence des documents préexistants est que l'on a besoin d'exécuter le programme de transformation sur toute la base de données XML pour garantir que tous les documents restent valides par rapport au nouveau schéma.

5.2 Adaptation des documents XML aux schémas

L'approche de transformation (ou adaptation) des documents XML à un schéma consiste à rendre un document (initialement valide par rapport à un schéma d) valide par rapport à un schéma cible d'. L'un des problèmes de la transformation est de mettre en correspondance les schémas, ce qui est difficile à faire sans l'intervention des utilisateurs administrateurs des applications concernées [BVPK04].

La méthode de transformation de documents s'énonce comme suit : soient un document XML X et un schéma d tel que $X \in L(d)$ (appelé schéma d'origine). Étant donné un schéma d' tel que $X \notin L(d')$ (appelé schéma cible), transformer X en X' tel que $X' \in L(d')$ et X' est le plus proche possible de X [KSR02, Lei03]. Ainsi, le défi de cette approche est de proposer une méthode de transformation des documents XML telle que la transformation modifie le moins possible les documents d'origine.

La majorité des travaux [BVPK04, SKR01, KLP02, Lei03] passe par les étapes suivantes :

1. Comparer le schéma d'origine d et le schéma cible d'.
2. Identifier les parties communes de d et d' (avec ou sans l'intervention de l'utilisateur).
3. Proposer un troisième schéma d_r tel que $L(d_r) \subseteq L(d')$.
4. Construire un programme de transformation F en utilisant d_r tel que, étant donné un document XML $X \in L(d)$, $F(X) \in L(d_r)$

Pour que la comparaison de d et d' soit efficace, les deux schémas doivent modéliser des applications des mêmes domaines [BVPK04, KLP02, RB01].

Le schéma d'origine d et le schéma cible d' peuvent être vus comme des arbres t_d et $t_{d'}$ [SKR01, KLP02, Lei03], respectivement. Les arbres schémas sont utilisés pour définir une correspondance entre leurs nœuds et construire le troisième schéma d_r. La construction de d_r est faite avec l'aide d'un utilisateur qui connaît l'application. La méthode est décrite comme suit [KLP02, Lei03] :

– D'abord, l'utilisateur fait l'association de chaque nœud feuille de t_d à un nœud feuille correspondant de $t_{d'}$ selon sa connaissance de la sémantique des schémas.
– Puis, la méthode fait l'association des nœuds internes de t_d et $t_{d'}$ de manière automatique, en se basant sur l'association faite par l'utilisateur.

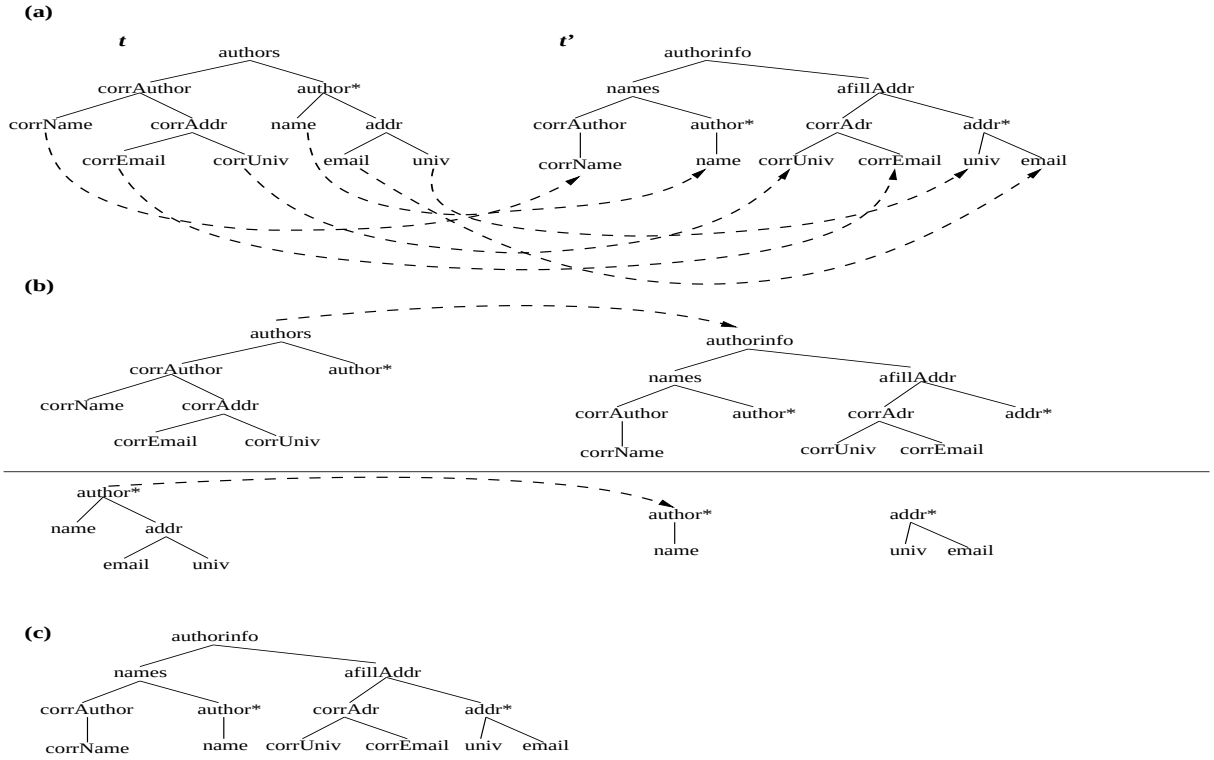


FIG. 5.1 – L'association des parties communes des arbres qui représentent des grammaires.

- Après l'association interne, l'arbre t_d est divisé en sous-arbres t_{ds}. La division se fait en fonction des associations faites dans les pas précédents. Pour chaque sous-arbre t_{ds} de nouveaux sous-arbres t_{dc} sont construits à partir de $t_{d'}$.
- Ensuite, l'utilisateur choisit le meilleur sous-arbre t_{dc} pour remplacer un sous-arbre t_{ds}.
- Pour terminer, les sous-arbres t_{dc} sont groupés, donnant l'arbre schéma t_{dr}.

Exemple 5.2.1 Étant donné les règles de la grammaire d'origine G_1 et de la grammaire cible G_2 suivantes[1] :

G_1 :
authors → **authors** (*corrAuthor author**)
corrAuthor → **corrAuthor** (*corrName corrAddr*)
author → **author** (*name addr*)
corrAddr → **corrAddr** (*corrEmail corrUniv*)
addr → **addr** (*email univ*)

G_2 :
authorinfo → **authorinfo** (*names affilAddr*)
affilAddr → **affilAddr** (*corrAddr addr**)
names → **names** (*corrAuthor author**)
corrAuthor → **corrAuthor** (*corrName*)
author → **author** (*name*)
corrAddr → **corrAddr** (*corrUniv corrEmail*)
addr → **addr** (*univ email*)

la figure 5.1 présente :

- **(a)** L'association faite par l'utilisateur entre les nœuds feuilles des arbres t et t' qui représentent les grammaires G_1 et G_2, respectivement.
- **(b)** Les sous-arbres t_{ds} (à gauche) construits à partir de G_1 et les sous-arbres candidats t_{dc} (à droite) construits à partir de G_1 et G_2. Les flèches montrent les choix faits par l'utilisateur entre les sous-arbres candidats possibles.
- **(c)** L'arbre t_{dr} construit à partir du choix fait par l'utilisateur.

Le programme de transformation est construit en utilisant l'arbre de la figure 5.1**(c)**. □

[1] Les règles de la forme $a \rightarrow \mathbf{a}(data)$ ont été omises.

Une autre approche d'adaptation de documents proposée dans [SKR01] est fondée sur le coût de transformer le schéma d'origine d en le schéma cible d'. Cette approche aussi prend en compte les schémas comme des arbres. Le coût est calculé par rapport au nombre d'opérations de mises à jour exécutées sur t_d pour obtenir $t_{d'}$. La méthode construit n séquences de mises à jour en utilisant une structure auxiliaire qui peut être un dictionnaire avec la sémantique de chaque élément des schémas, les domaines des éléments, etc. Le choix de la séquence à utiliser est basé sur le coût d'implanter la séquence sur t_d. La séquence la moins chère est utilisée pour construire un programme de transformation.

L'application des modèles conceptuels est aussi utilisée dans le cadre de la recherche de parties communes des schémas [BVPK04] pour adapter les documents à des schémas cibles. Les schémas concernés sont modélisés en deux niveaux :
(i) Niveau conceptuel qui modélise les schémas comme des objets réels en utilisant UML (Unified Model Language). Chaque déclaration d'élément est modélisée comme un objet de l'application du schéma, et
(ii) Niveau du schéma qui modélise les éléments des schémas en stockant leurs types (complexe ou simple), leur cardinalité, etc.
La figure 5.2 montre un exemple de modélisation conceptuelle des schémas : les rectangles représentent un élément et ses sous-éléments sont modélisés par relation de composition. Dans la figure 5.2 on remarque que *Personnel* est composé de *Enseignant*, c'est-à-dire, que l'élément *Enseignant* est un sous-élément de *Personnel*.

La construction d'un programme qui adapte le document d'origine au nouveau schéma est faite comme suit :

- D'abord, la méthode recherche dans le niveau conceptuel les similarités entre d et d' qui seront ensuite validées par l'utilisateur.
- Lorsque les similarités dans le niveau conceptuel sont validées, la méthode recherche les correspondances dans le niveau du schéma. Dans ce cas, il est aussi proposé des transformations dans les types de données. Par exemple, si l'élément *revenu* de d est du type réel et sa correspondance dans d' est du type entier, alors il faut traiter les valeurs réelles dans X pour les convertir en entier dans X'. Ce pas est aussi validé par l'utilisateur.
- Pour terminer, les correspondances engendrent des règles qui seront utilisées pour construire le programme qui transforme X en X' (avec $X \in L(d)$ et $X' \in L(d')$).

Contrairement aux méthodes proposées dans [SKR01, KLP02, Lei03], la méthode de [BVPK04] utilise comme langage de schéma le langage XML-Schema, et les types des données sont aussi pris en considération. Ainsi, la construction du programme de transformation doit aussi prendre en considération les types des données des éléments qui correspondent les uns aux autres.

La figure 5.2 montre la correspondance entre les modèles conceptuels des deux schémas. Remarquer que la correspondance *corr*00 fait correspondre les éléments *Personnel* et *Employe* et la correspondance *corr*01 fait correspondre les champs *prenom* et *nom* de *Personnel* au champ *Nom* de *Employe*. Remarquer que les correspondances peuvent avoir une liaison aussi, par exemple la correspondance *corr*01 est une composition de la correspondance *corr*00, c'est-à-dire, que *corr*00 est composé de plusieurs *corr*01. Cette information est prise en compte lors de la construction du programme de transformation.

Dans le cadre de l'évolution de schémas XML, la recherche de parties communes de schémas peut être modélisée comme suit : le schéma d'origine d est représenté par le schéma à mettre à jour et le schéma cible d' est représenté par le schéma mis à jour. L'énoncé du problème reste le même : faire en sorte que les documents d'origine soient valides par rapport au nouveau schéma en apportant le minimum de modifications sur eux et en utilisant un programme F qui transforme les documents valides par rapport à d en documents valides par rapport à d'. Le problème d'utiliser ces approches pour l'évolution des schémas est que le programme F doit être

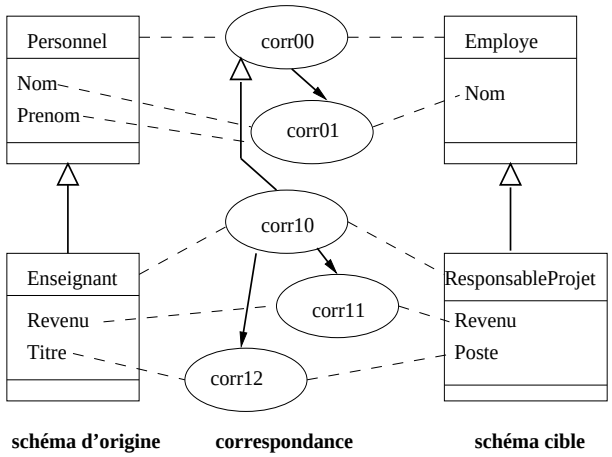


FIG. 5.2 – L'association des parties communes des schémas en utilisant un modèle conceptuel basé sur UML.

exécuté sur tous les documents pré-existants et valides dans la base de données XML ce qui apporte un coût supplémentaire dans le processus d'évolution des schémas.

5.3 Primitives de mises à jour des schémas XML

Cette section présente des approches qui proposent des primitives de mises jour sur les schémas. Comme toutes les propositions de mises à jour de schémas, les primitives doivent prendre en considération l'impact du changement. L'approche idéale est celle qui ne modifie pas les documents valides pré-existants dans la base de données XML [AE03], ce qui n'est pas toujours possible.

Les approches qui proposent des primitives de mises à jour des schémas sont toujours basées sur deux conditions [AE03, SKC+01] :

- Avant la mise à jour : l'opération de mise à jour doit respecter des pré-conditions avant d'être exécutée. Par exemple, un attribut a peut être associé à un élément e s'il n'existe pas un autre attribut a associé à e. Les pré-conditions sont vérifiées au niveau du schéma.
- Après la mise à jour : dès que le schéma change, tous les documents qui sont valides par rapport au schéma avant la mise à jour doivent être modifiés pour qu'ils soient valides par rapport au nouveau schéma. Cela est appelé propagation de mise à jour [Coo03]. Par exemple, si un élément obligatoire e est inséré dans le schéma, alors e doit être inséré dans tous les documents conformes au schéma avant la mise à jour.

On peut classer les primitives de mises à jour sur les schémas comme suit [AE03, SKC+01, Coo03] :

- Insertion, suppression et renommage d'éléments et d'attributs.
- Modification de l'opérateur d'un élément ($*$, $+$ et ?).
- Modification de la structure du modèle de contenu d'un élément. Par exemple, grouper les sous-éléments pour associer un nouvel opérateur au groupe.
- Modification du type d'un attribut.
- Remplacement d'un attribut par un élément et vice-versa.

L'exemple ci-dessous présente une de ces primitives de mise à jour et son effet sur les documents valides.

Exemple 5.3.1 Soient la définition suivante d'un élément qui modélise un *client* dans une DTD d :

<!ELEMENT Client (Nom, Adresse, Telephone)>
<!ATTLIST Client CliID #REQUIRED>

et l'opération de mise à jour [AE03] $ChangeElemToAttrib(Client, 1, NomAt)$ qui signifie changer, dans le modèle de contenu de $Client$, le premier sous-élément (dans ce cas Nom) par l'attribut $NomAt$. Les modifications à réaliser dans d et dans les documents conformes à d sont :

- L'attribut aura la cardinalité de l'élément à remplacer, dans cet exemple il est obligatoire.
- Pour chaque sous-élément <Nom> `text` </Nom> de $Client$ dans les documents valides, Nom est supprimé et la balise <$Client\ CliID$ = "001"> est remplacée par <$Client\ CliID$ = "001" $NomAt$ = "`text`">.
- La définition de l'élément $Client$ devient :
  <!ELEMENT Client (Adress, Telephone)>
  <!ATTLIST Client CliID #REQUIRED NomAt CDATA #REQUIRED>

Remarquer que le contenu du sous-élément Nom (`text`) devient le contenu de l'attribut $NomAt$. Cette méthode d'affectation n'est pas toujours valable : si le nouvel attribut est du type ID, alors la valeur `text` doit être unique dans tout le document et il ne peut pas y avoir d'espace à l'intérieur[2]. □

La propagation de la mise à jour des schémas dans les bases de données XML peut provoquer la perte de données dans les documents valides. Dans [AE03], les primitives ont la prémisse d'éviter la perte de données comme condition d'exécution, cependant dans [SKC+01], cette prémisse est allégée. Par exemple, si l'opérateur d'un sous-élément s d'un élément e change de $*$ en ?, l'approche dans [AE03] n'accepte la mise à jour que si, dans les documents XML valides, les éléments e ont, au maximum, un sous-élément s. Dans [SKC+01], les sous-éléments s sont supprimés sauf l'un d'entre eux.

Une autre façon de mettre à jour des schémas est de faire l'adaptation des schémas par rapport aux documents dans la base de données XML. Cette méthode cherche à adapter un schéma D à des documents qui ne sont pas valides par rapport à D. Dans ce contexte, D est mise à jour, donnant D', d'une façon que les documents valides par rapport à D restent valides par rapport à D'. Elle diffère des méthodes d'adaptation des documents aux schémas présentées dans la section 5.2 car l'objet adapté est le schéma et non les documents.

Dans ce contexte les documents dans la base sont classés en deux groupes : les documents valides Gr_v et les documents invalides Gr_i. La méthode s'énonce comme suit [BGMT02] :
(i) la base de données stocke les schémas et les documents conformes aux schémas, ces documents sont dans le group Gr_v,
(ii) de nouveaux documents arrivent dans la base de données : les documents qui sont conformes aux schémas de la base sont classés dans Gr_v, les documents qui ne sont pas conformes sont classés dans Gr_i,
(iii) pendant la vérification de classification, les documents de Gr_i sont associés au schéma qui leur convient le mieux. Pour chaque document X de Gr_i et pour chaque schéma D auquel X est associé, une structure auxiliaire est associée à toutes les règles $a \rightarrow \mathbf{a}\ (r_a)$ de D qui ne modélisent pas les éléments $\mathbf{a}$ de X. Le processus d'association des structures auxiliaires aux règles du schéma est appelé *phase d'enregistrement*, et
(iv) Dès que le group Gr_i a un nombre n de documents (où n est défini dans l'application), la *phase d'évolution* est déclenchée. Dans cette phase, les informations auxiliaires stockées dans la phase d'enregistrement sont utilisées pour changer les schémas pour que les documents dans Gr_i

[2]Les chaînes de caractères des attributs du type ID ne peuvent pas contenir des espaces (cela est valable pour les attributs du type IDREF).

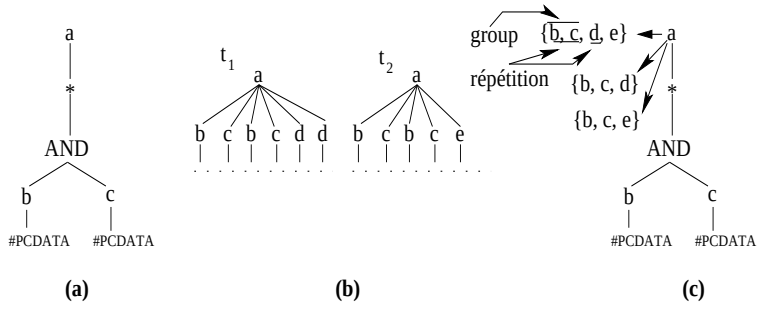


FIG. 5.3 – La phase d'enregistrement de l'adéquation d'un schéma.

soient conformes. S'il existe des documents dans Gr_i pour lesquels l'adéquation du schéma n'est pas possible, ces documents sont ignorés. L'adéquation est mesurée par rapport au nombre de mises à jour à faire dans les schémas.

La figure 5.3 montre les pas de la phase d'enregistrement :

- **(a)** le modèle de contenu d'origine de l'élément a (où AND signifie que le fils gauche doit être suivi par le fils droit, c'est-à-dire, la concaténation), noté *mod*. Il est représenté par un arbre. La règle qui représente *mod* est $a \rightarrow \mathbf{a}\ (b\ c)^*$.
- **(b)** deux morceaux des arbres t_1 et t_2 qui ne respectent pas *mod*.
- **(c)** le modèle de contenu avec la structure auxiliaire attachée. Elle est composée de :
  (i) Un ensemble S_e qui contient tous les sous-éléments d'un élément a qui n'est pas valide par rapport à *mod*. Cet ensemble prend en compte tous les documents (dans l'exemple les arbres t_1 et t_2). Remarquer que des informations concernant la répétition (comme d), ou la formation groupe (comme b et c) sont aussi représentées.
  (ii) Pour chaque élément a d'un document non-conforme, l'ensemble s_e des sous-éléments de a est construit.
  Dans cette figure, l'ensemble S_e est $\{b, c, d, e\}$ et pour t_1 l'ensemble $s_e^{t_1} = \{b, c, d\}$ et pour t_2 l'ensemble $s_e^{t_2} = \{b, c, e\}$.

La phase d'évolution est déclenchée lorsque le nombre de documents dans Gr_i atteint le seuil défini par l'application. À ce moment, de nouveaux modèles de contenu sont proposés à partir des structures auxiliaires. Ces nouveaux modèles doivent modéliser les documents dans Gr_v ainsi que les documents dans Gr_i.

Lorsque les structures auxiliaires sont construites, il est nécessaire de déterminer la fréquence de cas où présence d'un élément implique la présence d'autres éléments. Cette fréquence est calculée en utilisant les règles d'association. Les règles d'association sont extraites des structures auxiliaires comme suit [BGMT02] :

- Pour chaque modèle de contenu dans lequel la structure auxiliaire n'est pas vide, l'ensemble S_e est utilisé pour mettre à jour les ensembles s_e : tous les éléments e dans S_e qui n'appartiennent pas à s_e sont insérés dans s_e sous la forme $\overline{e}$. Les éléments $\overline{e}$ représentent les éléments manquants dans s_e. Par exemple, pour la figure 5.3, les ensembles $s_e^{t_1} = \{b, c, d\}$ et $s_e^{t_2} = \{b, c, e\}$ de a sont transformés, en utilisant $S_e = \{b, c, d, e\}$, ce qui donne $s_e^{t_1\prime} = \{b, c, d, \overline{e}\}$ et $s_e^{t_2\prime} = \{b, c, e, \overline{d}\}$.
- Pour tous les ensembles créés dans le pas précédent, des règles d'association sont construites. Par exemple, étant donné les ensembles $s_e^{t_1\prime}$ et $s_e^{t_2\prime}$, les règles d'association sont calculées pour les éléments dans les ensembles, *e.g.*, $c \rightarrow b$, $b \rightarrow c$, $c \rightarrow b\ d$, $b \rightarrow e\ \overline{d}$, etc. Ensuite, pour chaque règle créée, le support et la confiance sont calculés.
  On dit que la règle $X \rightarrow Y$ a pour support s dans les ensembles si $s\%$ des ensembles contient $X \cup Y$, et pour confiance c si $c\%$ des ensembles qui contiennent X contiennent aussi Y. Dans notre exemple, la règle $c \rightarrow b$ a le support 1 (c'est-à-dire, 100%) car $\{b, c\}$

apparaissent dans $s_e^{t_1\prime}$ et $s_e^{t_2\prime}$. La confiance est aussi 1 car tous les ensembles qui contiennent b contiennent aussi c. Les règles qui ont la confiance maximale pour le groupe calculé sont retenues.

Le but de construire les règles d'association $X \rightarrow Y$ est de vérifier si la présence de X implique la présence de Y ou, dans le cas de $X \rightarrow \overline{Y}$, si la présence de X implique l'absence de Y.

Les pas ci-dessus construisent un ensemble de règles d'association $\mathcal{R}$. En effet, dans notre exemple, les règles retenues[3] sont $a \rightarrow b$, $b \rightarrow a$, $d \rightarrow \overline{e}$ et $\overline{e} \rightarrow d$. À partir de $\mathcal{R}$, des règles d'heuristique sont appliquées. Ces règles sont des règles qui proposent la construction des modèles de contenus à partir de $\mathcal{R}$. Il existe des règles pour : la concaténation, l'union, l'ajout de l'opérateur $*$ dans un élément ou un groupe, entre autres.

Par exemple, s'il existe deux règles $x \rightarrow y$ et $y \rightarrow x$ dans $\mathcal{R}$, alors les éléments x et y sont concaténés , c'est-à-dire, ils sont groupés avec l'opérateur AND (l'expression régulière devient $x\ y$). S'il existe deux règles $x \rightarrow \overline{y}$ et $\overline{y} \rightarrow x$, alors les éléments x et y sont groupés de manière disjointe, c'est-à-dire, ils sont groupés avec l'opérateur OR (l'expression régulière devient $x|y$).

Après avoir appliqué les règles heuristiques, les modèles de contenu d'origine sont mis à jour et les documents dans Gr_i sont déplacés vers l'ensemble Gr_v. L'exemple 5.3.2 montre les pas de la phase d'évolution.

Exemple 5.3.2 Soit le contexte de la figure 5.3 la représentation de la phase d'enregistrement. Soit $S_e = \{b, c, d, e\}$ l'ensemble dans lequel b et c font partie d'un groupe et le groupe se répète, et d est un élément optionnel qui se répète. Soient $s_e^{t_1\prime} = \{b, c, d, \overline{e}\}$ et $s_e^{t_2\prime} = \{b, c, e, \overline{d}\}$ les ensembles où les règles d'association seront cherchées. Les pas de la phase d'évolution sont les suivants :

- L'ensemble $\mathcal{R} = \{b \rightarrow c, c \rightarrow b, d \rightarrow \overline{e}, \overline{e} \rightarrow d\}$ est construit à partir de s_1 et s_2.
- Les règles heuristiques sont appliquées en utilisant $\mathcal{R}$ et S_e.
- Le modèle de contenu $(b\ c)^*$ est modifié résultant en $(bc)^*(d^+|e)$

Les étapes de l'application des règles heuristiques sont présentées dans la figure 5.4.
Les règles d'association $b \rightarrow c$ et $c \rightarrow b$ font concaténer les éléments a et b. Les éléments b et c se répètent, alors ils sont transformés en $(b\ c)^*$ (sous-arbre (1) de la figure 5.4), l'opérateur * est hérité du schéma d'origine.
Les règles d'association $d \rightarrow \overline{e}$ et $\overline{e} \rightarrow d$ font l'union de ces deux éléments. L'élément d peut se répéter alors ils sont transformés[4] en $(d^+|e)$ (sous-arbre (2) de la figure 5.4).
Comme il n'existe plus d'éléments dans S_e, les sous-groupes construits sont concaténés, cela produit $(b\ c)^*(d^+|e)$ (sous-arbre (3) de la figure 5.4).
Le sous-arbre (4) de la figure 5.4 montre le nouveau modèle de contenu de l'élément a. □

Dans [BGMT02], les auteurs ne présentent pas la complexité de la recherche des règles d'association à partir des ensembles des sous-éléments et non plus le coût de construire et maintenir les structures auxiliaires. De plus, il existe treize types d'heuristiques à appliquer et elles peuvent se combiner en donnant plus de 40 combinaisons. Cela peut poser des problèmes à la méthode si on travaille avec des documents de taille importante.

L'approche présentée dans cette section peut être utilisée dans le cadre de l'évolution de schéma. En effet, elle est similaire à la contribution proposée dans cette thèse : la mise à jour qui transforme un document valide en invalide déclenche la mise à jour du schéma. Soient d un schéma et X un document valide par rapport à d (*i.e.*, $X \in Gr_v$). Si X est mis à jour, donnant X' et $X' \notin L(d)$, alors X est supprimé de Gr_v et X' est classé dans Gr_i. L'approche dans [BGMT02] peut être appliquée et le schéma d peut être mis à jour, donnant d' et $Gr_v \cup \{X'\} \subseteq L(d')$. Le

[3] Il existe des autres règles mais on ne les présentera par pour simplifier l'explication de la méthode.
[4] L'élément d est aussi optionnel, mais il est décoré avec le $^+$ car il apparaît dans un groupe disjoint.

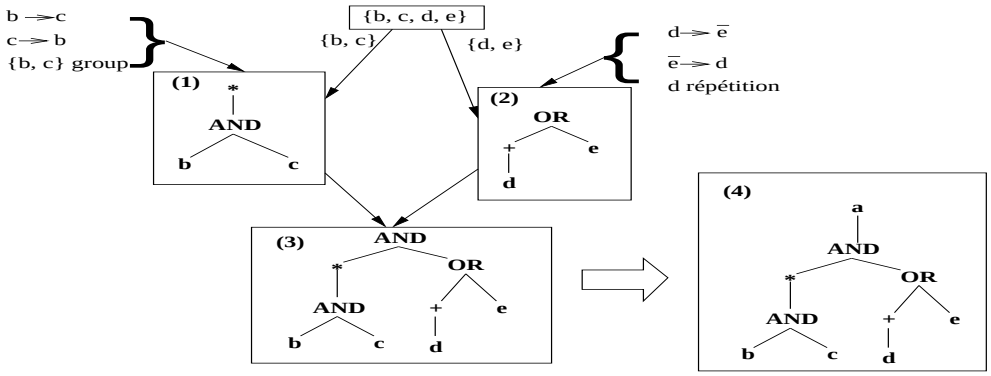


FIG. 5.4 – L'application de la phase d'évolution.

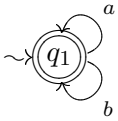
seul problème à résoudre est de définir le seuil n qui déclenche la phase d'évolution : il se peut qu'une seule mise à jour ne peut pas donner d'informations suffisantes pour la recherche des règles d'associations.

5.4 L'apprentissage des automates dans le cadre de l'évolution des schémas

Dans cette section, nous présentons le problème de l'apprentissage des automates, connu aussi comme l'inférence grammaticale (régulière). Nous montrons comment l'apprentissage peut être utilisé pour l'évolution des schémas XML. Nous nous intéressons à l'apprentissage des automates d'états finis (grammaire régulière) car les schémas XML sont modélisés par des expressions régulières.

L'inférence grammaticale est définie comme suit [PH00] : étant donné un ensemble fini d'exemples positifs (mots du langage), noté S^+, et un ensemble fini (possiblement vide) d'exemples négatifs (mots n'appartenant pas au langage), noté S^-, identifier une grammaire G^* qui est équivalente[5] à la grammaire cible G. Dans ce contexte, le choix de représentation de la grammaire cible joue un rôle important. Une grammaire régulière peut être représentée de façon équivalente par un ensemble de règles de production, par une expression régulière, par un automate d'états finis déterministe, ou par un automate d'états finis non-déterministe [HMU01]. La majorité des travaux sur l'inférence de grammaire utilise les automate d'états finis (DFA) déterministes comme la représentation de la grammaire régulière cible pour les raisons suivantes [DMV94] :

- Ils sont une représentation simple et facile à comprendre.
- Il existe plusieurs algorithmes (efficaces) pour exécuter plusieurs opérations sur les DFA (*e.g.*, faire la minimisation, tester l'équivalence entre deux automates, etc.).
- Il existe un seul DFA minimal qui représente une grammaire régulière.

Par exemple, étant donné $\Sigma = \{a, b\}$, $S^+ = \{ab, abab\}$ et $S^- = \{aab, abb\}$, une expression régulière qui décrit ce langage inférée à partir de S^+ et S^- peut être $E = (ab)^+$ car $S^+ \subset L(E)$ et $\forall e \in S^-$, $e \notin L(E)$. Remarquer que si l'ensemble S^- n'est pas donné, l'inférence peut inférer l'expression $E' = (a|b)^*$ construite à partir de l'automate : 

. Dans ce cas, tous les mots sur Σ appartiennent à $L(E')$ et donc, nous avons une sur-généralisation du langage appris [Gol67].

[5] Deux grammaires G_1 et G_2 sont équivalentes si leurs langages sont exactement les mêmes.

Les problèmes d'inférence grammaticale sont étudiés dans le cadre formel de la théorie de l'apprentissage automatique (*Computational Learning Theory*). Dans ce cadre, différents modèles théoriques de l'apprentissage ont été proposés. Ces modèles cherchent à définir formellement la généralisation à partir d'exemples, et proposent donc des critères d'identification des automates recherchés. L'identification de l'apprentissage est classée en deux critères : *l'identification à la limite* et *l'identification PAC* (*Problably Approximatly Correct*).

L'identification à la limite a été proposé par Gold [Gol67]. L'apprentissage, dans ce cas, doit être exact, c'est-à-dire, la grammaire apprise doit générer tous les mots du langage et aucun mot qui n'appartient pas au langage. Dans [Gol67], l'auteur montre que l'identification à la limite d'une grammaire régulière est possible seulement si l'apprenti a les deux ensembles (non-vides) d'exemples S^+ et S^-. Cependant, il est possible d'identifier des sous-classes de la grammaire régulière en utilisant le critère de l'identification à la limite et en n'utilisant que des exemples positifs : les classes de langages réversibles [Ang82]. Un langage L est 0-réversible si le miroir de l' automate déterministe M_L qui accepte L est aussi déterministe. On dit que M_L est 0-réversible. La notion s'étend à la k-réversibilité : le miroir est déterministe à horizon k, *i.e.*, après avoir lu k symboles.

Théorème 5.4.1 [Ang82] *Décider qu'un langage (donné sous forme d'automate quelconque) est k-réversible pour un certain k est décidable en temps polynomial.*

L'identification PAC a été proposée par Valiant [Val84] et elle relâche les contraintes de la présence d'exemples négatifs, néanmoins l'identification n'est pas exacte : il existe une probabilité de certitude. Ainsi, la grammaire inférée peut ne pas décrire tous les mots qui appartiennent au langage ainsi qu'elle peut décrire des mots qui n'appartiennent pas au langage.

Dans le cadre de l'apprentissage à la limite, l'un des algorithmes qui est la base de nombreux autres algorithmes est le RPNI (*Regular Positive and Negative Inference*) [OG92]. Nous allons montrer une étude de cas en essayant de faire un rapport entre l'algorithme RPNI (et une variation incrémentale de RPNI appelé RPNI2 - *Regular Positive and Negative Incremental Inference* [Dup96, DM98]) avec le problème d'évolution des schémas XML.

L'algorithme RPNI effectue un parcours en largeur dans l'arbre accepteur de préfixes PTA (*Prefix Tree Aceptor*) construit à partir de S^+ et trouve une solution optimale au problème du plus petit automate fini déterministe compatible avec S^+ et S^-. Ainsi, l'algorithme RPNI peut être résumé comme suit :
Entrée : Un échantillon (exemples positifs S^+ et négatifs S^-)

1. Construction de l'arbre préfixe.
2. Parcours en largeur de l'automate :
   (a) Fusion de (q1, q2) s'il n'existe rien qui l'empêche (*e.g.*, l'automate résultant accepte un mot dans S^-).
   (b) Déterminisation de l'automate construit.

Sortie : Un automate déterministe (le but).

L'exemple suivant présente des pas de la construction d'un automate en utilisant l'algorithme RPNI.

Exemple 5.4.1 Soient $S^+ = \{b, aa, aaaa\}$ et $S^- = \{\epsilon, a, aaa, baa\}$ les exemples positifs et négatifs d'un langage, respectivement. La figure 5.5 montre les étapes suivantes :

- **(a)** L'arbre accepteur de préfixes de S^+, représenté par un automate d'états finis.
- **(b)** Les états 1 et 0 fusionnés. Cependant l'automate accepte un exemple négatif : a. Cette fusion est rejetée.
- **(c)** Les états 2 et 0 fusionnés. Cependant l'automate accepte un exemple négatif : ϵ. Cette fusion est rejetée.

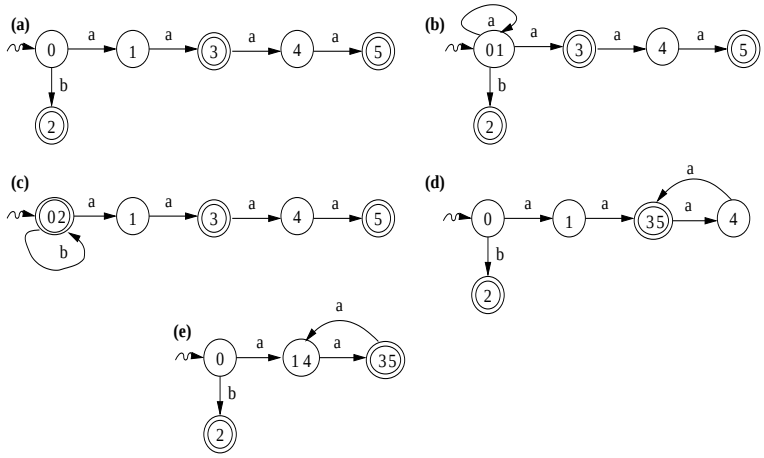


FIG. 5.5 – Pas de la construction d'un automate d'états finis en utilisant RPNI.

- **(d)** Les états 5 et 3 fusionnés. Aucun mot de S^- n'est accepté et ainsi, la fusion est acceptée.
- **(e)** Les états 4 et 1 fusionnés et l'automate résultant est l'automate (minimal) qui accepte S^+ et rejette S^-.

L'expression régulière construite à partir de l'automate de la figure 5.5 **(e)** est $E = b|(aa)^+$ et $S^+ \subset L(E)$ et $\forall e \in S^-$, $e \notin L(E)$. □

L'algorithme RPNI peut retourner un automate minimal si l'échantillon positif S^+ est structurellement complet relativement à un automate déterministe M. Pour cela [DM98] :

- Chaque transition de M est utilisée pour l'acceptation d'au moins un mot de S^+.
- Chaque état final de M est l'état d'acceptation pour au moins un mot de S^+.

L'ensemble S^+ de l'exemple 5.4.1 est structurellement complet relativement à l'automate de la figure 5.5**(e)**.

L'algorithme RPNI2 constitue l'extension incrémentale de l'algorithme RPNI. L'algorithme RPNI, à partir des ensembles d'exemples, construit la grammaire cible. Dès que les ensembles sont lus, l'apprentissage termine. Dans le cadre de l'algorithme RPNI2, le contexte change : de nouveaux exemples peuvent arriver après la construction de la grammaire cible et, dans ce cas, la grammaire peut changer.

L'approche utilisée par RPNI2 peut être résumée comme suit :

- Soit M_R l'automate construit de la façon standard. Soit $x = uv$ le nouvel exemple à traiter.
- Si x est un exemple positif et $x \notin L(M_R)$ ou si x est un exemple négatif et $x \in L(M_R)$, alors un nouvel automate est proposé :
  - Positif : Le préfixe le plus long u de x est accepté par M_R est vérifié. Pour les symboles dans v de nouveaux états sont insérés dans M_R, donnant M'_R, à partir du dernier état atteint avec le préfixe u. L'algorithme RPNI peut être alors appliqué sur M'_R.
  - Négatif : D'abord, on cherche dans quel état d'acceptation q_a le mot x arrive ($q_a \in F$ de M_R). À partir de q_a, un processus de fission (un processus inverse de la fusion d'états) est fait. La fission est exécutée sur les états fusionnés de façon inverse à la fusion. La fission s'arrête lorsque le mot x n'est plus accepté par le nouvel automate M'_R. L'algorithme RPNI peut être appliqué sur M'_R.

Remarquer qu'il faut stocker l'ordre de fusion des états pour appliquer l'approche proposée par l'algorithme RPNI2.

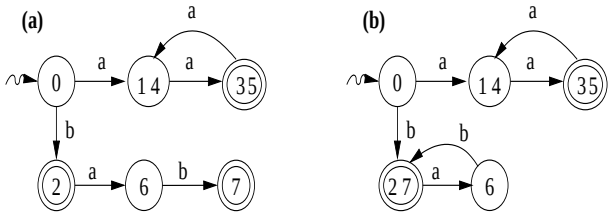


FIG. 5.6 – L'automate construit en utilisant l'algorithme RPNI2 à partir des automates de la figure 5.5.

Exemple 5.4.2 Soient M_R l'automate de la figure 5.5(e) et $w = bab$ le nouvel exemple positif que M_R ne reconnaît pas. La construction du nouvel automate est faite comme suit :

- L'état atteint par le plus long préfixe de w est 2 (préfixe b).
- De nouveaux états sont ajoutés à M_R (figure 5.6(a)) à partir de l'état 2 en utilisant le suffixe ab.
- L'algorithme RPNI est exécuté sur ce nouvel automate ce qui résulte en l'automate de la figure 5.6(b).

L'expression régulière résultante est $b(ab)^*|(aa)^+$ qui ne décrit pas les mots dans S^-. □

Maintenant, on peut présenter comment un algorithme d'inférence grammaticale peut être appliqué dans le cadre de l'évolution des schémas, plus précisément, l'adaptation du schéma à un document invalide. On suppose qu'un document X n'est pas valide car les sous-éléments d'un élément e de X ne respectent pas la contrainte imposée par le modèle de contenu de e. En d'autres termes, le mot résultant de la concaténation des sous-éléments de e n'appartient pas au langage décrit par l'expression régulière E qui modélise le modèle de contenu de e. Dans ce contexte, on suppose aussi que :

- Il faut faire évoluer le modèle de contenu de e, c'est-à-dire, E.
- Un automate d'états finis M_E est construit à partir de E pour reconnaître le langage décrit par E.
- Le mot w' est la concaténation des sous-éléments de e et $w' \notin L(E)$
- Il faut construire une nouvelle expression régulière E' qui décrit $L(E)$ et w', c'est-à-dire, w' devient un exemple positif pour l'automate M_E.

On peut traduire le contexte ci-dessus dans le cadre de l'inférence grammaticale comme suit : Un automate M_E existe et il doit reconnaître un nouvel exemple w' et aussi les mots reconnus auparavant. Remarquer que, en utilisant l'inférence grammaticale, la méthode d'évolution des schémas maintient la cohérence de la base de données XML sans modifier les documents valides préexistants.

Comme on a déjà un automate de départ, nous essayons d'utiliser l'algorithme RPNI2 pour le problème de l'évolution des schémas.

L'un des problèmes d'utiliser les méthodes d'inférence grammaticale est de trouver un ensemble d'exemples négatifs. On peut résoudre ce problème en proposant un ensemble d'exemples négatifs synthétique construit à partir des parcours dans M_E.

Comme le nouvel exemple donné à l'algorithme RPNI2 sera toujours un exemple positif, on n'a pas besoin d'avoir l'ordre de fusion des états à partir du PTA.

Soit S^- l'ensemble d'exemple négatifs construit à partir de M_E. On exécute l'algorithme RPNI2 ayant comme entrée l'automate d'origine M_E, le nouvel exemple positif w' et l'ensemble des exemples négatifs S^-. L'automate résultant sera celui qui accepte les mots acceptés par M_E et le mot w' (et peut-être des autres mots qui ne sont pas dans S^-).

À partir de l'exemple 5.4.1, supposons que l'expression régulière inférée $(b|(aa)^+)$ modélise le modèle de contenu d'une DTD :

```
<!ELEMENT Publication (Rapport|(Annee, Article)+)>
```

où $b = Rapport$, $a = Annee$ et $a = Article$.
Un document X est mis à jour donnant X' et le mot de l'un des éléments *Publication* de X' a changé de *Rapport* en *Rapport Annee Rapport*, c'est-à-dire que les sous-éléments *Annee* et *Rapport* ont été insérés. En utilisant les pas de l'exemple 5.4.2, on trouve une nouvelle expression régulière qui modélise tous les documents valides préexistantes dans la base de données et X'. Le nouveau modèle de contenu de *Publication* basé sur $b(ab)^*|(aa)^+$ est

```
<!ELEMENT Publication (Rapport,(Annee, Rapport)*|(Annee, Article)+)>
```

où $b = Rapport$, $a = Annee$, $b = Rapport$, $a = Annee$ et $a = Article$.

L'inconvénient d'utiliser l'apprentissage d'automates dans ce contexte est la construction de l'ensemble d'exemples négatifs : il faut compter la complexité de construction et le nombre d'exemples négatifs trouvés. Pour que les exemples négatifs soient utiles dans le processus de l'inférence, ils doivent être caractéristiques, *i.e.*, tous états distincts de l'automate cible soient distingués à l'aide d'exemples négatifs.

Pour éviter de construire l'ensemble synthétique d'exemples négatifs, on pourrait utiliser les algorithmes qui ne prennent pas en considération les exemples négatifs. Cependant, comme montré dans [Gol67], ces algorithmes n'apprennent pas l'ensemble complet des classes de langages réguliers.

Chapitre 6

Conclusion

Dans cette thèse, nous présentons (*i*) une méthode de validation des documents XML qui s'appuie sur les automates d'arbres qui prend en considération les attributs des documents et (*ii*) une méthode pour guider l'évolution des schémas XML qui préserve la validité des documents valides préexistants sans les modifier. De ce fait, nous sommes intéressé par deux aspects du langage XML : la validation de documents et la mise à jour de schémas. Dans ce contexte, nous avons présenté des domaines de recherche différents mais tous deux connectés à notre contribution, à savoir, les approches pour la validation de documents XML qui s'appuient sur des automates (chapitre 4) et les approches qui adaptent les documents à un schéma ou un schéma aux documents (chapitre 5).

Dans le chapitre 4, nous avons d'abord montré les méthodes de validation qui prennent en considération les documents XML vus sous la forme de chaînes caractères et dans ce cas, les automates d'états finis peuvent être utilisés dans le processus de validation. Bien que l'utilisation des automates d'états finis dans ce cadre soit importante principalement s'il existe des contraintes de mémoire centrale, ils ne peuvent pas être utilisés pour tous les documents valides. Si les schémas sont récursifs, l'application des automates d'états finis est réduite. Pour résoudre cela, on peut se servir d'autres types d'automates, par exemple, les automates à piles mais, dans ce cas, les contraintes de mémoire centrale sont allégées. Une autre manière de résoudre le problème des schémas récursifs par rapport aux automates d'états finis qui n'a pas été présentée dans ce chapitre est de transformer le schéma récursif non-reconnaissable pour qu'il devienne récursif reconnaissable [SV02].

La plupart des méthodes de validation s'appuient sur les automates d'arbres, comme le montre la section 4.2. Cela est naturel car les documents XML sont naturellement des arbres et leurs schémas peuvent être modélisés par des automates d'arbres. La logique peut aussi être utilisées pour valider les documents XML vus comme des arbres, cependant nous avons décidé de ne pas introduire les approches basées sur la logique, car elles ne sont pas directement liées à l'approche proposée dans cette thèse.

Parmi les approches de validation vues dans la section 4.2, seule l'approche proposée dans [MH03] prend en considération les contraintes structurelles des attributs. Les auteurs proposent de valider les modèles de contenu des éléments (sous-éléments et attributs) par des automates d'états finis. En effet, les attributs n'imposent pas d'ordre dans les modèles de contenus, et dans ce cas, leur modélisation par des expressions régulières est exponentielle [Kil99] et ainsi, l'utilisation des automates d'états finis n'est pas idéale. L'approche de [MH03], en utilisant la déclaration de non-existence d'attributs, propose le traitement des attributs et des éléments dans la même expression régulière. Néanmoins, leur méthode est laborieuse car à chaque validation d'un nœud de l'arbre XML, l'automate d'états finis correspondant doit être modifié pour traiter

les attributs.

Dans l'approche proposée dans cette thèse, nous prenons aussi (comme dans [MH03]) en considération les attributs mais ils sont considérés comme un ensemble, ainsi, la vérification des attributs est faite par des tests d'appartenance et la vérification des éléments par des automates d'états finis.

Nous avons présenté aussi les validations incrémentales. Ce type de validation est utilisée dans le cadre de mises à jour de documents XML et elle est utile puisque l'on évite de revalider tout l'arbre XML après la mise à jour d'un sous-arbre.

Dans le chapitre 5, nous avons présenté différentes méthodes qui adaptent soit les documents à un schéma, soit un schéma aux documents. Pour les méthodes présentés, nous avons montré une liaison entre ces méthodes et le besoin de faire évoluer les schémas XML.

Des approches présentées, seulement les approches de [AE03, Coo03, SKC$^+$01] travaillent directement avec les mises à jour exécutées dans les schémas par des utilisateurs. Ces auteurs proposent un ensemble de primitives de mises à jours qui est complet et correct. La majorité des ces primitives a la désavantage de ne pas maintenir la cohérence de la base de données XML sans modifier les documents préexistants.

Par ailleurs, l'approche proposée dans [BGMT02] maintient la validité de la base, toutefois, il peut exister des documents qui ne seront jamais conforme à un schéma, ils resteront toujours dans le groupe Gr_i, car ils peuvent être très "différents" des documents déjà existants et valides.

L'utilisation des approches qui sont basées sur la transformation des documents en utilisant soit la vérification de typage [KSS03, MBPS05, Suc02b, Via03], soit l'adaptation des documents aux schémas [BVPK04, SKR01, KLP02, Lei03] ont l'inconvénient de modifier les documents préexistants et ces modification peut amener à la perte de données.

Pour terminer, nous avons montré l'inférence grammaticale dans le cadre d'évolution des schémas. Cette méthode a l'avantage de maintenir la cohérence des documents XML valides préexistants sans les modifier. La désavantage de l'inférence grammaticale est que les modèles de contenu des schémas résultants peuvent au fur et mesure devenir plus complexe. Par exemple, dans la section 5.4, l'exemple présenté montre que un mot b qui appartenait à $L(b|(aa)^+)$ est mis à jour, donnant bab et, en conséquence, l'expression régulière a été modifié en $b(ab)^*|(aa)^+$ pour décrire le nouveau mot. Si des nouveaux mots sont présentés comme des exemples positifs (*e.g.*, *babcb*, *aabb*, etc), la sémantique de l'expression régulière d'origine peut ne plus être comprise par un être humain (*e.g.*, l'utilisateur responsable par l'application).

Troisième partie

Contribution

Chapitre 7

Validation de documents XML en considérant les éléments et les attributs

Pour valider des documents XML par rapport aux contraintes (structurelles) imposées sur les éléments et les attributs nous proposons une extension des automates d'arbres [BDHL03].

Notre méthode de validation considère un schéma représenté par un automate d'arbre résultant de la traduction d'une DTD. Dans ce cas, les contraintes suivantes doivent être respectées:

- Les expressions régulières qui apparaissent dans les règles de transitions doivent être non ambiguës[1] (voir définition 3.3.4).
- Chaque nom d'élément n'ayant qu'une définition (modèle de contenu), un seul état peut être associé à chaque nœud de l'arbre. C'est-à-dire que l'automate d'arbre représente une grammaire régulière d'arbres locale.

Rappelons les principes généraux de la validation XML par automate d'arbres:

- Les documents XML sont vus comme des arbres étiquetés d'arité non bornée ayant différents types de nœuds: données, élément et attribut.
- Des contraintes structurelles sur les attributs et les éléments sont imposées par un schéma. Nous utilisons les DTD comme langage de schéma et nous transformons chaque DTD d en un automate d'arbres étendu $\mathcal{A}$.
- L'exécution de $\mathcal{A}$ sur un arbre XML revient à vérifier si les contraintes sur les éléments et les attributs sont respectées. La vérification est faite par un seul parcours ascendant du document. Si toutes les contraintes sont satisfaites, alors l'exécution est dite réussie.

Bien qu'il existe plusieurs propositions pour la validation des documents XML par rapport à un langage de schéma [Chi00, MLM01, SV02], la majorité de ces propositions ne prennent pas en considération tous les détails du problème comme, par exemple, la validation structurelle des attributs. L'automate d'arbre proposé dans ce chapitre diffère de l'automate d'arbre décrit dans [BW98b, Nev99] dans la forme des règles de transitions. Les contraintes imposées par les attributs sont testées de deux façons différentes :

- Le premier test ne prend pas en considération les valeurs des attributs, il vérifie seulement si les attributs déclarés pour un élément e apparaissent comme fils de e, optionnels ou obligatoires (c'est une vérification d'appartenance à un ensemble).
- Le deuxième test prend en considération les valeurs de certains types d'attributs. Nous

[1] Cette restriction est préconisée par le *W3C* [BPSM98].

vérifions ici l'unicité des attributs du type ID et si les valeurs des attributs du type $IDREF$ et $IDREFS$ correspondent à des valeurs des attributs du type ID.

7.1 L'automate d'arbre ascendant étendu

Les automates d'arbres présentés dans la définition 3.2.1 sont étendus de façon à prendre en considération deux types de nœuds distincts dans un arbre XML: ceux qui sont traités comme membres d'un ensemble (que nous appelons le *groupe d'ensembles*) et ceux qui sont traités comme partie d'une séquence (que nous appelons le *groupe de séquences*). La définition ci-dessous formalise cet automate.

Définition 7.1.1 - Automate d'arbre non borné ascendant non déterministe étendu (ENFTA)[2]: Un automate ENFTA sur l'alphabet Σ est un quadruplet

$$\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$$

où Q est un ensemble d'états, $Q_f \subseteq Q$ est l'ensemble d'étas finaux et Δ est un ensemble de règles de transition de la forme

$$a, S, E \rightarrow q$$

où (i) $a \in \Sigma$; (ii) S est un couple composé de deux ensembles disjoints $< S_{compulsory}, S_{optional} >$ (avec $S_{compulsory} \cup S_{optional} \subseteq Q$); (iii) E est une expression régulière sur Q et (iv) $q \in Q$. □

Ainsi, pour remonter dans l'arbre en associant un état q à chaque nœud visité dans une position p, notre automate fait les tests suivants:

1. Si p a des fils du type attribut (*i.e.*, fils dans le groupe d'ensemble) alors les états associés à ces fils doivent correspondre aux états définis dans le couple S. L'ensemble $S_{compulsory}$ représente les fils de p qui sont des attributs et qui doivent apparaître dans l'arbre. L'ensemble $S_{optionnel}$ représente les fils de p qui sont des attributs et qui peuvent ou non apparaître dans l'arbre.
2. Si p a des fils du type élément (*i.e.*, fils dans le groupe de séquences) alors l'automate d'états finis, construit à partir de E, doit reconnaître le mot d'état formé de la concaténation des états associés aux fils de p dans la séquence.

L'automate d'arbre accepte un arbre s'il existe une exécution réussie, *i.e.*, s'il est possible de parcourir l'arbre à partir de ses feuilles jusqu'à sa racine en associant des états aux nœuds de façon à ce que l'état final soit associé à la racine. La définition suivante formalise l'exécution de l'automate d'arbre.

Définition 7.1.2 - Exécution de $\mathcal{A}$ sur un arbre t: Soit t un arbre et $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un automate d'arbre ascendant étendu. *L'exécution de $\mathcal{A}$ sur t* est l'application r, telle que $dom(r) = dom(t)$, et r défini comme suit: pour chaque position p dont les fils sont dans les positions $p0, \ldots, p(n-1)$ (où $n \geq 0$), l'état q est associé à p (*i.e.*, $r(p) = q$) si toutes les conditions suivantes sont vérifiées:

1. $t(p) = a \in \Sigma$
2. Il existe une transition $a, S, E \rightarrow q_a$ dans Δ telle que E est une expression régulière représentée par l'automate d'états finis $M_E = (\Gamma, Q, \delta, s_0, F)$.
3. Les fils de p peuvent être classés selon les règles suivantes:
   (a) Soit un entier $0 \leq i \leq (n-1)$ indiquant le nombre de fils de p du type attribut.

[2]Extended Non-deterministic bottom-up Finite Tree Automaton

(b) Les positions $p0, \ldots, p(i-1)$ appartiennent au groupe d'ensemble $setG$ (possiblement vide).

(c) Les positions $pi, \ldots, p(n-1)$ appartiennent au groupe de séquence $seqG$ (possiblement vide).

(d) Chaque fils de p est soit un membre de $setG$ soit un membre de $seqG$. Si une position p n'a pas de fils (*i.e.*, $n = 0$) alors $setG$ et $seqG$ sont vides. Si $n > 0$ et $i = 0$ alors $setG$ est vide. De manière similaire, si $n > 0$ et $i = n$ alors $seqG$ est vide.

4. L'arbre d'exécution r est déjà défini pour les positions $p0, \ldots, p(n-1)$, c'est-à-dire, $r(p0) = q_0, \ldots, r(p(n-1)) = q_{n-1}$.
5. Soit $seqG$ est vide auquel cas E est également vide, soit le mot $q_0 \ldots q_{n-1}$ (*i.e.*, la concaténation des états associés aux positions dans $seqG$) est accepté par l'automate M_E.
6. Le couple S respecte les propriétés suivantes:
   (a) $S_{compulsory} \subseteq \{q_0, \ldots, q_{i-1}\}$ où $\{q_0, \ldots, q_{i-1}\}$ est l'ensemble des états $setG$ et
   (b) $\{q_0, \ldots, q_{i-1}\} \setminus S_{compulsory} \subseteq S_{optional}$.

Une exécution r est dite *réussie* si $r(\varepsilon)$ est un état final et un arbre t est *accepté* s'il existe une exécution réussie sur t. □

La définition 7.1.2 est une définition constructive de l'exécution de $\mathcal{A}$. Elle montre comment l'automate ascendant "travaille" sur un arbre, en associant des états aux fils d'un nœud p avant d'associer un état à p. Nous appelons *move*, l'opération d'association $r(p) = q$, notée aussi par $p \rightarrow_{\mathcal{A}} q$, ce qui signifie que l'état q est associé au nœud dans la position p. Soit $\rightarrow^*_{\mathcal{A}}$ la fermeture transitive et réflexive de $\rightarrow_{\mathcal{A}}$. L'exécution de $\mathcal{A}$ sur t est $t \rightarrow^*_{\mathcal{A}} r$ tel que $r(\varepsilon) \in Q_f$.

Pour faciliter la lecture du reste de ce chapitre nous répétons des figures déjà présentées dans la partie I de cette thèse. La figure 7.1 présente un morceau d'un document XML valide par rapport à la DTD de la figure 7.2. La figure 7.3 présente l'arbre t du document de la figure 7.1 et son arbre d'exécution r avec les positions de chaque nœud.

Exemple 7.1.1 Soit l'arbre XML de la figure 7.3. Nous considérons un automate d'arbre $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, où $\delta_1 : data, < \emptyset, \emptyset > , \emptyset \rightarrow q_{data}$ et $\delta_2 : Publication\ , < \emptyset, \emptyset >, q_{Sujet}\ (q_{Annee} {q_{Revue}}^+)^* \rightarrow q_{Publication}$ sont des règles de transitions dans Δ. En utilisant la transition δ_1 toutes les feuilles de t sont associées à l'état q_{data}.
Maintenant, considérons le nœud dans la position 02 pour décrire une opération *move* : supposons que les fils de 02 (les nœuds dans les positions 020, 021 et 022) sont associés aux états q_{Sujet}, q_{Annee} et q_{Revue}, respectivement. L'élément $Publication$ (figure 7.2) n'a pas d'attributs alors nous fixons la valeur de i à 0 (définition 7.1.2) et ainsi, $setG = \emptyset$ et $seqG = \{020, 021, 022\}$. D'après la contrainte de schéma exprimée par δ_2 : (i) $S_{compulsory} \subseteq setG$, (ii) $setG \setminus S_{compulsory} \subseteq S_{optional}$ et (iii) le mot d'état $q_{Sujet}\ q_{Annee}\ q_{Revue}$ est accepté par l'automate d'états finis associé à $q_{Sujet}\ (q_{Annee}\ {q_{Revue}}^+)^*$. Ainsi, l'état $q_{Publication}$ peut être associé à la position 02 dans r. □

Remarquer que la valeur de l'entier i est fixée selon les types de nœuds dans l'arbre t. Pour savoir si un nœud dans un arbre XML correspond à un attribut ou à un élément nous avons besoin de fonctions externes. Elles sont importantes dans la construction de notre algorithme de validation (comme le montre la section ci-dessous) car elles permettent de récupérer des informations additionnelles sur les nœuds d'un arbre. Dans ce contexte, étant donné un arbre t, une position $p \in dom(t)$ et un entier j, nous avons les fonctions externes suivantes:

- $children(t, p)$ fait correspondre à p l'ensemble de toutes les positions pj dans $dom(t)$. Nous appelons $feuilles$ de t toutes les positions telles que $children(p) = \emptyset$.
- $father(t, p)$ fait correspondre à p une position $u \in dom(t)$ telle que $uj = p$. Nous définissons $father(t, \varepsilon) = \varepsilon$.

```
<Universite Nom="Université de Moscou">
 <Laboratoire>
   <Nom> Informatique </Nom>
   <Chercheur CId="C001">
     <Nom> Victor M. Glushkov </Nom>
     <Titre> Professeur </Titre>
   </Chercheur>
   <Publication>
     <Sujet> Automates </Sujet>
     <Annee> 1961 </Annee>
     <Revue>
        <Nom> Revue Russe de Recherche </Nom>
        <TArticle idAuts="C001">
               Théorie Abstraite des Automates
        </TArticle>
     </Revue>
   </Publication>
 </Laboratoire>
</Universite>
```

FIG. 7.1 – Un document XML concernant les laboratoires de recherche d'une université.

```
<!DOCTYPE Universite [
 <!ELEMENT Universite (Laboratoire*)>
 <!ELEMENT Laboratoire (Nom,Chercheur*,Publication*)>
 <!ELEMENT Nom (#PCDATA)>
 <!ELEMENT Chercheur (Nom,Titre)>
 <!ATTLIST Chercheur CId  ID #REQUIRED>
 <!ELEMENT Publication (Sujet,(Annee,Revue+)*)>
 <!ELEMENT Sujet (#PCDATA)>
 <!ELEMENT Annee (#PCDATA)>
 <!ELEMENT Revue  (Nom,TArticle)>
 <!ELEMENT Titre (#PCDATA)>
 <!ELEMENT TArticle (#PCDATA)>
 <!ATTLIST TArticle IDAuts  IDREFS #REQUIRED>
]>
```

FIG. 7.2 – DTD qui modélise les laboratoires de recherche attachés à une université

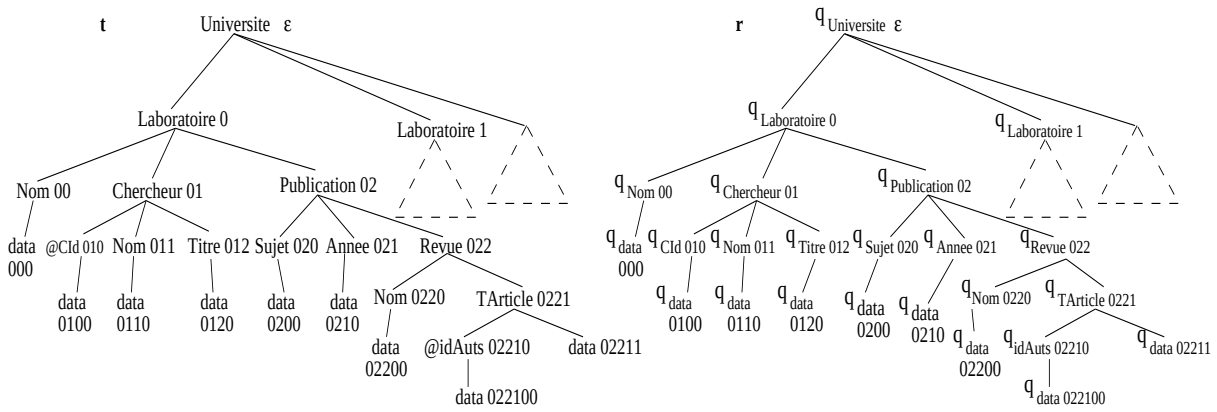


FIG. 7.3 – L'arbre XML avec ses positions et l'arbre d'exécution correspondant.

– $type(t, p)$ fait correspondre à p une valeur qui indique si p est un attribut, un élément ou un arbre vide. Elle est définie comme suit:

$$type(t,p) = \begin{cases} element & \text{si} \quad t(p) \in \Sigma_{ele} \cup \{data\} \\ attribute & \text{si} \quad t(p) \in \Sigma_{att} \\ emptytree & \text{si} \quad t(\varepsilon) = \lambda \end{cases}$$

– $value(t, p)$ est définie comme suit[3]:

$$value(t,p) = \begin{cases} setval \subset \mathbf{D} & \text{si} \quad type(t,p) = attribute \\ undefined & \text{autrement} \end{cases}$$

où $\mathbf{D}$ est un domaine infini récursivement énumérable.

Exemple 7.1.2 Considérons l'application des fonctions ci-dessus sur l'arbre t dans la figure 7.3 nous avons: $children(t, 0) = \{00, 01, 02\}$, $children(t, 000) = \emptyset$, $father(t, 02) = 0$, $type(t, 010) = attribute$, $type(t, 020) = element$, $value(t, 020) = undefined$, $value(t, 010) = \{\text{C001}\}$. □

7.2 Transformation d'une DTD en ENFTA

Nous présentons la définition de la transformation d'une DTD (selon la définition 3.3.1 dans le chapitre 3) en un automate d'arbre étendu (selon la définition 7.1.1). L'algorithme qui implante cette définition et un exemple de transformation sont présentés dans l'annexe A.

Définition 7.2.1 - Construction d'un ENFTA à partir d'une DTD: Étant donné une DTD d, nous définissons l'automate d'arbre d'arité non bornée non déterministe étendu $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ associé à d par les règles suivantes:

1. L'alphabet Σ est $\Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$ (voir définition 3.3.2).
2. L'ensemble d'états Q est construit comme suit: pour chaque symbole $a \in \Sigma$ nous ajoutons q_a dans Q, *i.e.*, $Q = \{q_a \mid a \in \Sigma\}$.
3. Q_f est un singleton contenant l'état $q_{firstEle}$ qui correspond à l'élément le plus externe de d (*i.e.*, l'élément $firstEle$).
4. L'ensemble de règles de transitions Δ se construit de la manière suivante :
   (a) Pour chaque déclaration d'élément dans d ayant la forme générale <!ELEMENT *eleName modèle-de-contenu*> construire une nouvelle règle de transition $\delta : a, S, E \rightarrow q_a$ où $S =< \emptyset, \emptyset >$ et où a, E et q_a sont définis comme suit:

[3]Notons que nous ne nous intéressons pas aux valeurs des éléments (*i.e.*, PCDATA).

i. Le symbole a est le nom de l'élément $eleName$ et q_a est le nom de l'état correspondant à a.

ii. Si le modèle de contenu est $regExp$ alors E est l'expression régulière obtenue en remplaçant tous les symboles s dans $regExp$ par q_s.

iii. Si le modèle de contenu est #PCDATA alors E est q_{data}.

iv. Si le modèle de contenu est EMPTY alors E est $\emptyset$.

v. Si le modèle de contenu est MIXTE alors nous utilisons les items *ii* et *iii* ci-dessus pour construire E.

(b) Pour chaque déclaration d'attribut dans d ayant la forme générale <!ATTLIST $eleName$ $attSet$> faire:

– Pour chaque n-uplet[4] $\nu[att\text{-}name, att\text{-}kind, att\text{-}status]$ dans $attSet$:

i. Construire une nouvelle règle de transition $\delta : att, < \emptyset, \emptyset >, q_{data} \rightarrow q_{att}$ telle que $att = \nu(att\text{-}name)$.

ii. Mettre à jour le couple S dans la règle de transition $eleName, S, E \rightarrow q_{eleName}$ de la façon suivante:

si $\nu(att\text{-}status)$= #REQUIRED

alors $S_{compulsory} = S_{compulsory} \cup \{q_{\nu(att-name)}\}$

sinon $S_{optional} = S_{optional} \cup \{q_{\nu(att-name)}\}$.

(c) Ajouter la règle de transition $data, < \emptyset, \emptyset >, \emptyset \rightarrow q_{data}$ à Δ. □

Exemple 7.2.1 Soit <!ELEMENT Chercheur (Nom, Titre)>
<!ATTLIST Chercheur CId ID #REQUIRED >
une déclaration d'élément et ses attributs dans une DTD quelconque. La règle de transition construite pour $Chercheur$ selon la définition 7.2.1 est:

$$Chercheur, < \{q_{CId}\}, \emptyset >, q_{Nom}\ q_{Titre} \rightarrow q_{Chercheur}$$

□

Remarquer dans la définition 7.2.1 que nous définissons un morphisme entre les ensembles Σ et Q comme une application $h : \Sigma \rightarrow Q$ qui fait correspondre les étiquettes $a \in \Sigma$ à ses états $q_a \in Q$ telle que si $abc \in \Sigma^*$ alors $h(abc) \in Q^*$.

Le modèle de contenu d'un élément e est une expression régulière E. Pour vérifier si le modèle de contenu de e est respecté il faut vérifier si le mot d'états w construit à partir des fils du nœud e dans l'arbre XML appartient au langage régulier décrit par E. De ce fait, il faut construire un automate d'états finis pour chaque expression régulière des règles de transitions. Il existe plusieurs algorithmes pour construire un automate à partir d'une expression régulière [HMU01], nous avons choisi l'algorithme proposé par V. M. Glushkov [CZ00] qui construit un automate d'états finis avec $n + 1$ états sans transitions vides (où n est le nombre de symboles dans l'expression régulière). Nous décrivons brièvement les automates de Glushkov, *i.e.*, les automates d'états finis construits en utilisant l'algorithme de Glushkov.

La méthode proposée par Glushkov s'appuie sur les positions des symboles dans l'expression régulière. Afin de préciser la position des symboles dans une expression régulière E, les symboles sont indicés dans l'ordre de lecture. Par exemple, à partir de l'expression $a?\ b\ a$, on obtient l'expression indicée $a_1?\ b_2\ a_3$; les indices sont appelés *positions*. L'ensemble des positions pour

[4]Nous utilisons la notation de base de données relationnelles proposée dans [AHV95]. Considérant que ν est un n-uplet sur V, nous écrivons $\nu(A)$ pour dénoter la valeur d'un attribut A de V dans ν.

une expression E est noté $Pos(E)$. Si F est une sous-expression de E, $Pos_E(F)$ dénote le sous-ensemble des positions[5] de E qui sont positions de F. À toute expression E est associée bijectivement une expression indicée $\overline{E}$. L'alphabet de $\overline{E}$ est noté par $\sigma = \{\alpha_1, \ldots, \alpha_n\}$, où $n = |Pos(E)|$. Sans perte de généralité, nous écrivons $\overline{E}$ pour représenter l'expression E avec des indices ou avec seulement les indices sans les symboles. Par exemple, étant donné $E = a?\ b\ a$, nous écrivons $\overline{E}$ pour $a_1?\ b_2\ a_3$ ou simplement $1?\ 2\ 3$

Afin de construire un automate d'états finis non déterministe (AFN) reconnaissant $L(E)$, Glushkov a établi trois fonctions que l'on peut définir de la façon suivante sur l'expression indicée $\overline{E}$:

Définition 7.2.2 - L'ensemble des positions initiales des mots de $L(E)$ ($First(\overline{E})$): [ZPC97] $First(\overline{E}) = \{x \in Pos(E) \mid \exists u \in \sigma^* : \alpha_x u \in L(\overline{E})\}$ □

Définition 7.2.3 - L'ensemble des positions finales des mots de $L(E)$ ($Last(\overline{E})$): [ZPC97] $Last(\overline{E}) = \{x \in Pos(E) \mid \exists u \in \sigma^* : u\alpha_x \in L(\overline{E})\}$ □

Définition 7.2.4 - L'ensemble des positions qui suivent immédiatement la position x dans E ($Follow(\overline{E}, x)$): [ZPC97] $Follow(\overline{E}, x) = \{y \in Pos(E) \mid \exists v \in \sigma^*, \exists w \in \sigma^* : v\alpha_x\alpha_y w \in L(\overline{E})\}$ □

Ces fonctions permettent de définir l'automate $\overline{M}$ qui reconnaît $L(\overline{E})$. Ainsi, nous avons $\overline{M} = (Q, \alpha, \Delta, 0, F)$ où:

1. $Q = Pos(E) \cup \{0\}$
2. $\forall x \in First(\overline{E})$, $\delta(0, \alpha_x) = \{x\}$
3. $\forall x \in Pos(E)$, $\forall y \in Follow(\overline{E}, x)$, $\overline{\delta}(x, \alpha_y) = \{y\}$
4. $F = Last(\overline{E})$ ou $F = Last(\overline{E}) \cup \{0\}$ si $\varepsilon \in L(E)$.

À partir de $\overline{M}$, l'automate $M_E = (Q, \Gamma, \Delta, 0, F)$ est construit de la façon suivante: les arcs de M_E se déduisent de ceux de $\overline{M}$ en remplaçant leur étiquette α_x par $\chi(x)$. L'application χ fait correspondre à chaque position de $Pos(E)$ le symbole de l'alphabet de E qui apparaît à cette position dans E. Nous avons donc [ZPC97]

$$\Delta = \{\delta(q, a) = q' \mid q \in Q \wedge q' \in Q, a \in \Gamma \wedge (\exists \alpha_x)((q, \alpha_x) = q' \in \Delta) \wedge \chi(x) = a\}$$

L'exemple suivant montre les étapes de construction d'un automate de Glushkov présentées ci-dessus.

Exemple 7.2.2 Étant donné l'expression régulière $E = a\ (b\ c^+)^* d$. L'expression indicée correspondant est $\overline{E} = a_1\ (b_2\ c_3^+)^* d_4$ (ou simplement $\overline{E} = 1\ (2\ 3^+)^* 4$).
Nous avons : $Pos(E) = \{1, 2, 3, 4\}$, $first(\overline{E}) = \{1\}$, $last(\overline{E}) = \{4\}$, $follow(\overline{E}, 1) = \{2, 4\}$, $follow(\overline{E}, 2) = \{3\}$, $follow(\overline{E}, 3) = \{2, 3, 4\}$ et $follow(\overline{E}, 4) = \emptyset$. À partir des fonctions ci-dessus, nous avons $\overline{M} = \{\{0, 1, 2, 3, 4\},\ \{0, 1, 2, 3, 4\}, \{\delta(0, 1) = 1,\ \delta(1, 2) = 2,\ \delta(1, 4) = 4,\ \delta(2, 3) = 2,\ \delta(3, 2) = 2,\ \delta(3, 3) = 3,\ \delta(3, 4) = 4\}\ ,0,\ \{4\}\}$. Remplaçant les position x par $\chi(x)$, nous avons $M_E = \{\{0, 1, 2, 3, 4\},\ \{a, b, c, d\}, \{\delta(0, a) = 1,\ \delta(1, b) = 2,\ \delta(1, d) = 4,\ \delta(2, c) = 2,\ \delta(3, b) = 2,\ \delta(3, c) = 3,\ \delta(3, d) = 4\}\ ,0,\ \{4\}\}$. □

Comme déjà expliqué dans le chapitre 3, les modèles de contenu des éléments dans une DTD doivent être non ambigus. L'automate d'arbre $\mathcal{A}$ (définition 7.2.1) est construit à partir d'une DTD et nous avons donc des expressions régulières non ambiguës dans les règles de transitions. Dans ce cas, les automates de Glushkov construits à partir des expressions régulières des règles de transitions de $\mathcal{A}$ sont déterministes [BW98a].

Il faut remarquer l'ajout d'un symbole (#) à la fin de chaque expression régulière E avant de construire l'automate de Glushkov correspondant. De même, avant de vérifier si un mot w

[5] Pour $E = F + G$ et $E = F\ G$, $Pos_E(F) \cap Pos_E(G) = \emptyset$; pour E^*, $Pos(E^*) = Pos(E)$

appartient au langage E, nous ajoutons aussi # à w. Dans le cas, les automates de Glushkov ont la propriété d'avoir toujours un seul état initial et un seul état final [CZ00]

On note que la conversion d'une expression régulière en un automate de Glushkov est un algorithme qui s'exécute en temps $O(n^2)$ (où n est le nombre de symboles dans l'expression régulière) [BK93, CZ00, ZPC97]. Ainsi, l'algorithme de traduction d'une DTD d en un automate d'arbre $\mathcal{A}$ (selon définition 7.2.1) a un temps d'exécution en $O(|\Sigma_{ele}| + |\Sigma_{att}|) \times t_{Glushkov}$, où $t_{Glushkov}$ est $O(n^2)$. En effet, l'algorithme de transformation parcourt toute la DTD et à chaque fois qu'il trouve une déclaration $<!ELEMENT \ldots >$ ou $<!ATTLIST \ldots >$, une règle de transition δ est créée et pour l'expression régulière E dans δ, un automate de Glushkov est construit.

D'autre part, pendant la traduction de d en $\mathcal{A}$, une relation T qui stocke les propriétés des attributs de d est aussi construite. T est utilisée pour guider le stockage des valeurs des attributs ID et IDREF/IDREFS vérifiées pendant le processus de validation.

Définition 7.2.5 - Table de propriétés des attributs T: Étant donné une DTD d, la *Table de propriétés des attributs* T est construite pendant le processus de construction de $\mathcal{A}$ comme suit:

- pour chaque déclaration d'attribut dans d de la forme <!ATTLIST *eleName attSet*> faire:
  - pour chaque n-uplet ν[*att-name*, *att-kind*, *att-status*] dans *attSet* ajouter le triplet ν_1[*ele*, *att-name*, *att-kind*] dans T, avec:
    $\nu_1(ele) = eleName$,
    $\nu_1(att\text{-}name) = \nu(att\text{-}name)$ et
    $\nu_1(att\text{-}kind) = \nu(att\text{-}kind)$. □

Dans la suite nous présentons les propriétés de l'automate d'arbre étendu de la définition 7.1.1.

Théorème 7.2.1 *Étant donné une DTD d, l'automate d'arbre $\mathcal{A}$ construit à partir de d en utilisant la définition 7.2.1 est toujours un automate d'arbre régulier fini déterministe ascendant.*

Preuve: (*i*) $\mathcal{A}$ est déterministe car d représente une grammaire régulière d'arbres locale. Par conséquent, il n'existe pas deux règles de transition qui peuvent associer à un même nœud deux états différents.
(*ii*) $\mathcal{A}$ est ascendant car l'exécution commence par les feuilles et remonte vers la racine en associant un état à chaque nœud visité (définition 7.1.2).
(*iii*) $\mathcal{A}$ est fini car les ensembles d'états, de symboles (alphabet) et de règles de transitions sont finis puisque la définition de d est finie (définition 7.2.1).
(*iv*) $\mathcal{A}$ est régulier car la règle de transition $a, S, E \rightarrow q$ peut être réduite à $a, E \rightarrow q$ en traitant les attributs comme membres du groupe de séquences (dans ce cas on transforme les ensembles $S_{compulsory}$ et $S_{optional}$ en expressions régulières selon l'approche proposée dans [Kil99]). □

Nous finissons cette section en prouvant qu'une DTD d et l'automate d'arbre étendu $\mathcal{A}$ correspondant expriment le même ensemble d'arbres.

Théorème 7.2.2 *Étant donné une DTD d et l'automate d'arbre $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ construit selon la définition 7.1.1, le langage généré par d est équivalent au langage reconnu par $\mathcal{A}$, i.e., $L(d) = L(\mathcal{A})$.*

Preuve: voir annexe B □

```
string function move (p, A, t) {
// Calcule l'état à associer à la position p de t en utilisant A.
// Entrées: la position p, l'automate d'arbre A et l'arbre t.
// Sortie : l'état q ∈ Q à associer au nœud dans la position p de t ou
//          q_error si un état valide n'est pas trouvé.
   Let children(t,p) = {p0,...,p(n-1)}, n ≥ 0
   set_childrenStates = ∅
   seq_childrenStates = ε
   // les états déjà calculés pour {p0,...,p(n-1)} sont placés
   // dans l'ensemble d'attributs et dans la séquence d'éléments
   for i from 0 to n-1 do {
     if (type(t,pi) = attribute) {
       set_childrenStates = set_childrenStates ∪ {r(pi)}
     }
     else {
       seq_childrenStates = seq_childrenStates · r(pi)
     }
   }
   a = t(p)
   Let δ be the transition rule a,S,E → q_a
   Let S be the tuple < S_compulsory, S_optional >
   if ((the automaton M_E defined by E recognizes seq_childrenStates))
   and (S_compulsory ⊆ set_childrenStates)
   and (set_childrenStates \ S_compulsory ⊆ S_optional)) {
     return q_a
   else {
     return q_error
   } }
```

FIG. 7.4 – Fonction *move*.

7.3 Les algorithmes de validation

Dans cette section nous présentons notre algorithme de validation XML. Pour cela, nous considérons : (*i*) l'arbre XML t, (*ii*) l'automate d'arbre étendu $\mathcal{A}$ et (*iii*) la table (relation) T construits à partir d'une DTD d. Le processus de validation est représenté par la fonction $valid(\mathcal{A}, T, t)$. Les trois algorithmes suivants font partie du processus de validation: (*i*) *move* est responsable de l'association d'un état q à une position p (figure 7.4), (*ii*) *run* implante la définition 7.1.1 en se servant de *move* (figure 7.5), et (*iii*) *valid*, en se servant de *run*, vérifie si un automate $\mathcal{A}$ accepte un arbre t (figure 7.6).

Nous remarquons que la procédure *run* utilise deux autres procédures qui sont détaillées ci-dessous :

1. La fonction *noOneGreater* retourne la position la plus grande dans C selon la relation de préfixes (définition 2.2.1 dans le chapitre 2). Par exemple, si $C = \{00, 001\}$ alors la position 001 est choisie avant la position 00. Cependant, si $C = \{000, 001, 010, 1\}$ ou $C = \{00, 010, 1\}$ alors n'importe quelle position peut être choisie car elles ne peuvent pas être comparées selon leurs préfixes.
2. La procédure $idIdref$ est chargée de stocker dans deux tables les valeurs des attributs. Les deux tables sont V_{ID} et V_{IDREFS} (elles sont traitées comme variables globales). La

```
procedure run (A, T, t, r) {
// Exécute l'automate d'arbre A sur t.
// Entrées: l'automate d'arbre A, la table des types d'attributs T et l'arbre t.
// Sortie : Une exécution r de A sur t, i.e., l'arbre r tel que
//          dom(t) = dom(r) et les nœuds de r sont étiquetés par Q.
   C = fr(t) //fr(t) retourne les positions frontières de t
   pos = noOneGreater(C)
   while (C ≠ ∅) do {
     if (type(t, pos) = attribute) {
       idIdref(T, t, pos) // stocke les valeurs ID/IDREF(S)
     }
     r(pos) =move(pos, A, t)
     C = C \ {pos}
     if (pos ≠ ε) {
       C = C ∪ {father(t, pos)}
       pos = noOneGreater(C)
     }
   }
}
```

FIG. 7.5 – Procédure *run*.

procédure a comme paramètres d'entrées la table T (définition 7.2.5), l'arbre t et la position p et exécute le pas suivant :

- Si le nœud dans la position p est un attribut alors il a seulement un fils dans la position $p0$. La procédure $idIdrefs$ vérifie le type du nœud p et ajoute la valeur retournée par $value(t, p0)$ à V_{ID} (respectivement à V_{IDREFS}) si le type de p est ID (respectivement $IDREF$ ou $IDREFS$).

La fonction *valid* exécute d'abord l'automate d'arbre sur un arbre XML t et ensuite, vérifie la conformité des attributs de type ID et $IDREF(S)$ (la vérification est faite en utilisant les tables V_{ID} et V_{IDREFS}).

```
boolean function valid (A, T, t) {
// Vérifie si l'arbre t est accepté par A et la conformité des attributs
// ID/IDREF(S).
// Entrées: l'automate d'arbre A, la table T et l'arbre t.
// Sortie : true si l'arbre est accepté et les valeurs dans V_ID et
//          V_IDREFS sont conformes aux spécifications de la DTD.
   run(A, T, t, r)
   if (r(ε) ∈ Q_f and checkIdIdref()=true) {
     return true
   }
   else {
     return false
   }
}
```

FIG. 7.6 – Fonction *valid*.

La fonction $checkIdIdref$ appelée lorsque l'automate d'arbre accepte l'arbre t par la fonction *valid*, exécute les pas suivants:

– Elle vérifie si chaque valeur dans V_{ID} est unique.
– Elle vérifie que toute les valeurs dans V_{IDREFS} ont une entrée dans V_{ID}.

Si les deux conditions ci-dessus sont respectées, $checkIdIdref$ retourne $true$ sinon elle retourne $false$. Ainsi, si la fonction run réussit et les valeurs des attributs du type $ID/IDREF(S)$ respectent leurs restrictions imposées alors nous concluons que l'arbre t est accepté par l'automate $\mathcal{A}$. En fait, selon le théorème 7.3.1 ci-dessous, nous pouvons aussi dire que le document XML représenté par t est valide par rapport à la DTD d à l'origine de $\mathcal{A}$. Le théorème 7.3.1 montre que notre méthode est complète et correcte. Autrement dit, nous prouvons que seuls les documents valides par rapport à la DTD d sont acceptés par la procédure de validation (correction) et, inversement, seuls les arbres acceptés par la procédure de validation correspondent aux documents valides par rapport à d (complétude).

Théorème 7.3.1 *Étant donné une DTD d, son automate d'arbre étendu $\mathcal{A}$ et sa table de propriétés des attributs T, soit X un document XML et t son arbre étiqueté par Σ. Le document XML X est conforme à d si et seulement si l'appel* `valid(`$\mathcal{A}$`,`T`,`t`)` *retourne* `true`.

Preuve: D'abord nous prouvons la correction de notre méthode de validation, c'est-à-dire, nous prouvons que si X est conforme à d alors la fonction $valid$ retourne $true$ pour les paramètres d'entrée t et $\mathcal{A}$.
Du fait que X est conforme à d, nous savons que (i) l'arbre XML associé à X appartient au langage d'arbre généré par d ($i.e.$, $t \in L(d)$) et (ii) les valeurs ID et $IDREF(S)$ respectent les propriétés d'unicité et d'existence, respectivement. D'après le théorème 7.2.2, si $t \in L(d)$ alors $t \in L(\mathcal{A})$ et ainsi, la procédure run (appelée dans la fonction $valid$) réussit ($i.e.$, $r(\varepsilon) \in Q_f$). La fonction $checkIdIdref$ retourne $true$ car l'unicité des attributs ID implique que les valeurs dans V_{ID} sont aussi uniques et l'existence de valeurs ID pour tous les attributs $IDREF(S)$ implique que pour toutes les valeurs dans V_{IDREFS}, il existe une entrée dans la table V_{ID}. Donc, la fonction $valid$ retourne $true$ lorsque X est conforme à d.
Maintenant, nous prouvons la complétude de notre méthode, c'est-à-dire, si la fonction $valid$ retourne $true$ avec t et $\mathcal{A}$ comme entrées alors X est conforme à d.
Du fait que $valid$ retourne $true$ lorsque t et $\mathcal{A}$ sont donnés comme paramètres nous savons que (i) $t \in L(\mathcal{A})$ et (ii) la fonction $checkIdIdref$ retourne $true$. D'après le théorème 7.2.2, il est clair que si $t \in L(\mathcal{A})$ alors $t \in L(d)$. Par la définition de la fonction $checkIdIdref$, nous savons que les restrictions imposées par les attributs du type $ID/IDREF(S)$ sont respectées. Donc, X (le document XML associé à t) est conforme à d. □

La complexité de la méthode de validation est en $O(n_e + n_a + c_i(c_i + c_r))$, où n_e est le nombre d'éléments, n_a est le nombre d'attributs dans X, c_i et c_r représentent le nombre des valeurs des types ID et IDREF/IDREFS, respectivement. En effet, les valeurs n_a et n_e représentent le nombre des nœuds visités pendant la validation (chaque nœud est visité une seule fois), et après avoir rempli les tables des valeurs ID et IDREF/IDREFS, il faut les parcourir pour vérifier s'il n'existe pas de doublons dans la table ID et si toutes les valeurs de la table des IDREF/IDREFS ont une valeur correspondante dans la table de ID.

La méthode ici présentée a été implantée en Java en utilisant le modèle d'arbre [LMM00] qui considère que l'arbre XML est stocké dans la mémoire vive et qu'il est accédé par une API (Application Programming Interface) telle que DOM [WHA+00]. Cependant, l'utilisation du modèle par événement (en utilisant une API telle que SAX [Meg00]) est aussi possible car ce modèle a le même comportement qu'un automate d'arbre ascendant puisque les événements de fermeture de balises sont d'abord déclenchés par les balises les plus internes, c'est-à-dire, les nœuds feuilles de la représentation arborescente.

Nous finissons cette section en présentant des expérimentations réalisées avec l'implantation

TAB. 7.1 – Comparaisons entre notre algorithme (fonction *valid*) et les API JAVA $JDOM$ et $JSAX$.

		Entrées		*Temps en secondes*		
Exp.	*nœuds*	*ID*	*IDREF(S)*	*valid*	*JDOM*	*JSAX*
1.	196	10	17	0,050	0,345	0,190
2.	962	53	114	0,110	0,791	0,201
3.	1.924	106	228	0,150	0,811	0,291
4.	7.696	424	912	0,561	0,891	0,711
5.	30.784	1.696	3.648	1,973	1,612	1,232
6.	70.955	3.097	8.419	5,328	2,824	2,063
7.	221.045	12.172	26.238	15,391	8,793	5,157
8.	442.090	24.344	52.476	97,596	68,778	10,265

d'un prototype de l'algorithme de validation. Ce prototype a été implanté avec le but de vérifier le comportement des procédures présentées dans cette section. Il a été une base pour l'implantation d'un validateur plus performante par notre groupe de recherche.

La table 7.1 présente les résultats de ces expérimentations qui ont été faites avec un ordinateur ayant un processeur Pentium III 866 Mhz et 192 Mo de mémoire vive sous Windows 2000.

Nous avons décidé de tester la performance du prototype par rapport au nombre de nœuds dans l'arbre XML au lieu d'utiliser la taille du fichier. Notre choix est justifié par le fait que la taille n'est pas directement liée au nombre de tests et le nombre d'expressions régulières à vérifier. Par exemple, un document XML de 1 megaoctets peut avoir seulement 10 nœuds car il peut avoir beaucoup de texte à l'intérieur.

Nous avons créé 8 fichiers XML qui sont conformes à la DTD de la figure 7.2 et dans lesquels les données sont synthétiques. La fonction *valid* qui implante notre méthode de validation a été comparée avec les API's de validation Java $JDOM$ et $JSAX$. L'API $JSAX$ est la plus performante pour les documents avec plus de 200.000 nœuds. En effet, la validation SAX est faite de façon séquentielle pendant la lecture du document XML car aucune autre structure auxiliaire n'est créée. Quant à lui, DOM construit un arbre dans la mémoire centrale pendant la validation et cela explique sa performance moins efficace par rapport à SAX. Le choix entre utiliser DOM et SAX est fait par rapport aux besoins d'une application : s'il est nécessaire parcourir des morceaux du document XML plusieurs fois, l'API DOM est idéale car en utilisant des algorithmes *breath-first* ou *deph-first* on peut faire des parcours efficaces dans l'arbre. Si le document n'est lu qu'une fois, alors l'API SAX est le meilleur choix car il n'existe pas de structures auxiliaires à créer. Comme *valid* a été implantée en utilisant l'API DOM, alors il est naturel que sa performance soit comparée avec $JDOM$.

La table 7.1 nous montre que, malgré le fait que le prototype a été implanté d'une façon simple, sa performance est intéressante par rapport aux API $JDOM$ et $JSAX$. Par exemple, considérons l'arbre avec 221.045 nœuds, 12.172 entrées de valeurs des attributs du type ID et 26.238 entrées de valeurs d'attributs de type IDREF/IDREFS. Dans ce contexte, *valid* a pris un peu plus de 15 secondes pour valider le document tandis que $JDOM$ a pris $8,793$ secondes et $JSAX$ a pris $5,345$ secondes. Nous tenons aussi à noter que l'implantation de la vérification d'unicité et d'existence des valeurs dans les tables V_{ID} et V_{IDREF} de *valid* est faite d'une façon simple : des parcours séquentiels dans les tables V_{ID} et V_{IDREF}. Ainsi, lorsque le nombre de valeurs ID et $IDREF/IDREFS$ augmente, *valid* baisse sa performance par rapport à $JDOM$ et SAX, dont nous n'avons pas trouvé l'approche de vérification d'unicité ID et d'existence des $IDREF/IDREFS$.

Dans ce chapitre, nous avons présenté une méthode pour valider des documents XML basée sur les automates d'arbres qui traite aussi bien les éléments que les attributs d'un document XML. Par ailleurs, la méthode traite aussi la validation de l'unicité (attributs ID) et d'existence (attributs IDREF/IDREFS) des attributs dans un document XML.

L'utilisation d'un langage de schéma qui s'appuie sur les expressions régulières pour définir les modèles de contenu des éléments nous amène à définir une méthode pour guider l'évolution de ce type de schéma. La méthode de guidage de l'évolution de schéma est fondée sur l'échec de l'automate d'états finis M_E associé à une expression régulière E lors de la vérification de l'appartenance d'un mot à $L(E)$. Cette méthode est l'objet du chapitre suivant dans lequel nous présentons cette évolution incrémentale des schémas XML.

Chapitre 8

Évolution incrémentale des schémas XML

Dans ce chapitre nous présentons une méthode pour guider l'évolution de schémas XML. Dans le chapitre 5 nous avons présenté des méthodes qui considèrent le changement des documents lorsque le schéma change, ainsi que des approches prenant en compte l'évolution des schémas eux-mêmes. Notre méthode est différente des méthodes présentées dans le chapitre 5 car l'évolution des schémas est déclenchée par la mise à jour des documents; les documents eux-mêmes demeurent inchangé suite à l'évolution de leur schéma [BDH+04a, BDH+04b].

8.1 L'évolution des schémas activée par la mise à jour des documents XML

Nous considérons un environnement d'échange de données fondé sur XML dans lequel existent aussi bien des utilisateurs ordinaires que des administrateurs. Dans ce contexte, nous nous intéressons aux mises à jour de documents XML qui satisfont des contraintes de schéma (*i.e.* des documents *valides*). Pendant la mise à jour d'un document XML valide, il faut vérifier si le document résultant est encore conforme aux contraintes imposées par le schéma. Si le document n'est plus valide par rapport au schéma, on dit que la mise à jour est invalide. *Les mises à jour invalides* peuvent être traitées de différentes manières, selon le type d'utilisateur qui les exécute. Celles exécutées par des utilisateurs ordinaires sont rejetées, alors que celles exécutées par des administrateurs peuvent être acceptées et ainsi amener à des corrections dans le schéma.

Les mises à jour de documents XML faites par des utilisateurs ordinaires sont traitées dans [BH03], où une méthode qui préserve incrémentalement la validité des documents est proposée. Dans ce travail, les auteurs ont proposé un ensemble de primitives de mise à jour qui respectent la validité des documents XML. Ces primitives sont présenté dans la section 4.3.

Dans cette thèse nous proposons une méthode d'aide à l'utilisateur expert dans le domaine d'une application mais pas forcément expert en informatique. Ainsi, nous supposons que cet utilisateur peut déclencher des modifications sur le schéma via une mise à jour sur un document XML (initialement valide par rapport à son schéma et invalide si la mise à jour est effectuée). Notre approche consiste à proposer des modifications sur le schéma de façon à rendre valide le document mis à jour. Cette méthode a les caractéristiques suivantes :

– Si une mise à jour d'un document XML viole les contraintes de schéma, des actions correctives sont prises pour rétablir la validité du document.

– Les documents auparavant valides restent valides par rapport au nouveau schéma.
– La méthode propose différentes options à l'administrateur. Les caractéristiques du schéma d'origine, du document et de la mise à jour demandée sont prises en considération pour construire chaque option. L'administrateur peut alors décider quel schéma adopter d'après sa connaissance sémantique des documents.

8.2 Comment modifier un schéma à partir d'une mise à jour : vision générale de notre approche

Les mises à jour sur un document sont traitées comme des modifications apportées à l'arbre correspondant au document. Par exemple, une requête d'insertion revient à une insertion d'un sous-arbre t' à une position p de t. Avant d'accepter l'insertion, il faut vérifier si le nouvel arbre respecte les contraintes définies par le schéma $\mathcal{A}$. Ces vérifications sont incrémentales : si on insère à une position p, il suffit de vérifier si la partie du document associée au père de p respecte encore le schéma [BH03]. Si le père de p n'est plus accepté par $\mathcal{A}$, nous pouvons proposer des modifications sur $\mathcal{A}$ à l'administrateur.

L'exemple suivant montre comment une requête d'insertion demandée par l'administrateur peut apporter des modifications dans le schéma.

Exemple 8.2.1 Étant donné l'arbre XML t, son arbre d'exécution r de la figure 7.3 et la règle de transition

$$Publication, <\emptyset,\emptyset>, q_{Sujet}\ (q_{Annee}\ (q_{Revue})^+)^* \rightarrow q_{Publication} \tag{8.1}$$

Supposons l'insertion d'un arbre t_a à la position 023 dans l'arbre t (figure 7.3). Supposons aussi que l'exécution de $\mathcal{A}$ sur t_a ait associé à la racine de l'arbre d'exécution r_a l'état $q_{Conference}$.

Le sous-arbre qui doit être vérifié par $\mathcal{A}$ est le sous-arbre dont la racine se trouve à la position 02 (le père de la position 023) : $\mathcal{A}$ doit associer à la position 02 l'état $q_{Publication}$ en utilisant la règle (8.1). Après l'insertion de t_a dans t, le mot d'état w construit par $\mathcal{A}$ pour les fils de 02 de type élément est $q_{Sujet}\ q_{Annee}\ q_{Revue}\ q_{Conference}$. Il est clair que $w \notin L(q_{Sujet}\ (q_{Annee}\ (q_{Revue})^+)^*)$. Ainsi, dans ce schéma, un article publié dans une conférence n'est pas une publication. Dans ce cas, cette insertion est dite invalide.

Considérons maintenant que l'utilisateur qui a exécuté la mise à jour est l'administrateur de l'application et qu'il veut que cette mise à jour soit considérée aussi comme une demande de mise à jour du schéma. Le but est de remplacer $\mathcal{A}$ par un nouvel automate $\mathcal{A}'$ avec les caractéristiques suivantes : (i) tous les documents XML qui respectaient $\mathcal{A}$ respectent $\mathcal{A}'$ et (ii) $\mathcal{A}'$ est différent de $\mathcal{A}$ seulement par rapport à l'expression régulière qui a été affectée par la mise à jour. Ainsi, l'évolution du schéma consiste à choisir la nouvelle expression régulière qui remplacera E dans (8.1). Il est évident que ce choix dépend des aspects sémantiques de l'application. Notre méthode propose des options à l'administrateur, qui sont obtenues en fonction de l'expression régulière d'origine et du mot w non reconnu. Dans cet exemple, les options sont les suivantes (triées selon le niveau auquel $q_{Conference}$ est inséré) :

1. Au même niveau que q_{Sujet}. Les *articles de conférence* doivent être organisés par *sujet* sans prendre en considération l'*année* de publication :
   (a) $q_{Sujet}\ (q_{Annee}\ (q_{Revue})^+)^* q_{Conference}?$: pour un *sujet* on peut avoir (au maximum) un article de conférence.
   (b) $q_{Sujet}\ (q_{Annee}\ (q_{Revue})^+)^* (q_{Conference})^*$: pour un *sujet* on peut avoir n articles ($n \geq 0$).
2. Au même niveau que q_{Annee}. Les articles de conférence seront aussi organisés par *année* mais ils doivent apparaître après les articles de revue :

(a) $q_{Sujet}\ (q_{Annee}\ (q_{Revue})^{+} q_{Conference}?)^{*}$: pour chaque année on peut avoir (au maximum) un article de conférence.
(b) $q_{Sujet}\ (q_{Annee}\ (q_{Revue})^{+} q_{Conference}{}^{*})^{*}$: pour chaque année on peut avoir n articles ($n \geq 0$).

3. Au même niveau que q_{Revue}. Les articles de conférence seront organisés comme les articles de revues :
(a) $q_{Sujet}(q_{Annee}(q_{Revue}\ q_{Conference}?)^{+})^{*}$: après un article de revue, on peut avoir (au maximum) un article de conférence.
(b) $q_{Sujet}(q_{Annee}(q_{Revue}\ q_{Conference}{}^{*})^{+})^{*}$: après un article de revue, on peut avoir n articles de conférence ($n \geq 0$).
(c) $q_{Sujet}(q_{Annee}(q_{Revue}|q_{Conference})^{+})^{*}$: les articles de revues et de conférences n'ont pas d'ordre mais ils sont toujours organisés par l'année de publication.

L'administrateur peut alors choisir la meilleure option pour son besoin. □

Notons que les expressions candidates présentées dans l'exemple 8.2.1 sont classées selon le niveau d'insertion du nouveau symbole dans l'expression régulière d'origine. Dans la suite de ce chapitre nous montrons comment il est possible de classer les options de cette manière.

Cet exemple montre également que les expressions candidates E' que nous calculons correspondent à des langages $L(E')$ plus généraux que $L(E) \cup \{w\}$ (w étant le mot non reconnu). Le but est, en effet, d'accepter toute une nouvelle famille de documents en essayant de prévoir les besoins de l'administrateur. La solution triviale $E|w$ a peu de chance d'intéresser l'administrateur (pas plus qu'une solution extrêmement générale).

Nous nous intéressons aux expressions régulières candidates E' qui sont aussi similaires que possible de l'expression d'origine E. Comme le montre l'exemple précédent, les candidats proposés ont seulement un nouveau symbole ajouté par rapport à l'expression d'origine. Cette condition nous aide à définir une notion de distance très simple, présentée dans la suite de ce chapitre.

Étant donné une règle de transition $a, S, E \rightarrow q$. Soit $w[0:n]$ (avec $n \geq 0$) un mot composé de la concaténation de $n+1$ états associés aux fils d'un nœud dans un arbre XML t valide (*i.e.*, $w \in L(E)$). Étant donné un symbole dans la position i de w, *i.e.*, $w[i]$ (avec $0 \leq i \leq n$), l'insertion (respectivement la suppression) d'un sous-arbre t_p à la position p dans t, correspond à la construction d'un nouveau mot d'état $w'[0:n+1]$ (respectivement $w'[0:n-1]$). Si w' n'appartient plus au langage de E alors la règle de transition $a, S, E \rightarrow q$ ne peut pas associer l'état q au nœud père de p. Ainsi, à partir d'une expression régulière E et d'un nouveau mot $w' \notin L(E)$ construit à partir d'un mot $w \in L(E)$, la méthode consiste à :

1. Construire l'automate d'états finis M_E reconnaissant $L(E)$.
2. Faire des modifications dans M_E pour obtenir de nouveaux automates M'_E qui acceptent w'.
3. Construire une nouvelle expression régulière E' associée à chaque automate M'_E.

Les nouvelles expressions régulières E' sont construites en respectant la structure imposée par E (*i.e.*, nombre de sous-expressions étoilées, symboles disjoint, etc) et les caractéristiques du nouveau mot. Les items 1 et 2 de cette méthode sont réalisés par un algorithme appelé `GREC` qui est une extension du processus de transformation d'un automate de Glushkov en une expression régulière présenté dans [CZ00].

8.3 Les fondements de notre méthode d'induction de schéma

Notre approche est fondée sur la méthode de transformation d'un automate de Glushkov en une expression régulière proposée par Caron et Ziadi [CZ00]. Dans cette section nous présentons brièvement cette méthode ainsi que les définitions concernant les automates de Glushkov.

8.3.1 Les propriétés des automates de Glushkov

Les automate d'états finis (AEF) peuvent être représentés sous forme graphique. Les états de l'automate sont les nœuds du graphe, et la relation de transition définit un ensemble d'arcs étiquetés. Chaque nœud est étiqueté avec le symbole de la transition correspondante. Dans la figure 8.1(a), nous présentons un automate qui reconnaît le langage de l'expression régulière[1] $E = (a(b|c)^*)^*\#$. Notons que cet automate est homogène, puisqu'il respecte les conditions imposées par la définition ci-dessous.

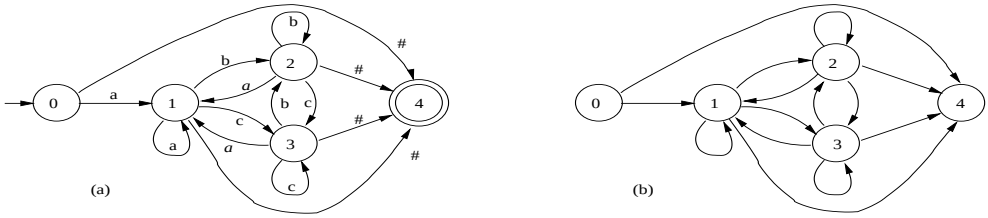


FIG. 8.1 – (a) L'automate pour $(a(b|c)^*)^*\#$ et (b) son graphe de Glushkov.

Définition 8.3.1 - Homogénéité [CZ00] : Soit $M_E = (Q, \Gamma, \Delta, q_0, F)$ un automate d'étas finis. M_E est dit *homogène* si $\forall p, q, r \in Q$, nous avons : $\exists a, b \in \Gamma$, $((p, a, q) \in \Delta \wedge (r, b, q) \in \Delta) \Rightarrow a = b$. En d'autres termes, un état est toujours atteint par le même symbole. □

Nous rappelons qu'un *graphe orienté* $G = (X, U)$ est une paire composée d'un ensemble de nœuds X et d'une relation U dans X. Chaque élément de U est appelé un *arc*. Un arc $u = (r, s)$ est une boucle si $r = s$. Étant donné un graphe G, un chemin est une séquence de nœuds $x_0, \ldots, x_n$ telle que, pour tout $0 \leq i < n$, l'arc (x_i, x_{i+1}) est dans G. Un chemin qui n'a pas d'arc est appelé trivial.

La figure 8.1(b) montre que, dans le cas d'un automate homogène, nous pouvons enlever les symboles étiquetant les arcs[2]. Nous appelons *graphe de Glushkov* le graphe ainsi obtenu à partir d'un automate de Glushkov.

Un graphe a un nœud *racine* (respectivement un nœud *anti-racine* s) r s'il existe un chemin de r à n'importe quel nœud (respectivement de n'importe quel nœud à s) dans le graphe. Un graphe est appelé *hamac* s'il a un nœud racine r et un nœud anti-racine s, tels que $r \neq s$.

Définition 8.3.2 - Orbite et orbite maximale [CZ00] : Soit $G = (X, U)$ un graphe. Un ensemble $\mathcal{O} \subseteq X$ est appelé une *orbite* si pour tout x et x' de $\mathcal{O}$, il existe un chemin non trivial de x à x'. Une orbite est *maximale* si pour tout nœud x qui appartient à $\mathcal{O}$ et pour tout nœud x' qui n'appartient pas à $\mathcal{O}$, il n'existe pas à la fois un chemin de x à x' et un chemin de x' à x. □

Notons qu'une orbite est maximale si elle n'est incluse dans aucune autre orbite et que les orbites maximales représentent les composantes fortement connexes ayant au moins un arc (*i.e.*, les chemins triviaux ne sont pas pris en compte).

[1] Rappelons que le symbole de fin # est toujours ajouté E pour construire l'automate d'états finis.

[2] Rappelons que l'application χ fait correspondre à chaque position de $Pos(E)$ le symbole de l'alphabet de E. Ainsi, pour un nœud $x \in X$, $\chi(x)$ représente l'étiquette des arcs entrants dans x.

Définition 8.3.3 - Porte d'entrée et de sortie d'une orbite [CZ00] : Soit $G = (X, U)$ un graphe. Soit $\mathcal{O}$ une orbite de G. L'ensemble $In(\mathcal{O})$=$\{x \in \mathcal{O} \mid \exists x' \in (X \setminus \mathcal{O}), (x', x) \in U\}$ dénote les *portes d'entrée* de $\mathcal{O}$ et $Out(\mathcal{O})$=$\{x \in \mathcal{O} \mid \exists x' \in (X \setminus \mathcal{O}), (x, x') \in U\}$ dénote les *portes de sortie* de $\mathcal{O}$. □

Définition 8.3.4 - Stabilité et stabilité forte [CZ00] : Soit $G = (X, U)$ un graphe. Soit $\mathcal{O}$ une orbite de G. L'orbite $\mathcal{O}$ est *stable* si pour tout $x \in Out(\mathcal{O})$ et pour tout $y \in In(\mathcal{O})$, l'arc (x, y) existe. Une orbite $\mathcal{O}$ est *fortement stable* si elle est stable et si, après la suppression des arcs $Out(\mathcal{O}) \times In(\mathcal{O})$, toutes les orbites résultantes sont fortement stables. □

Définition 8.3.5 - Transversalité et transversalité forte [CZ00] : Soit $G = (X, U)$ un graphe. Soit $\mathcal{O}$ une orbite de G. Une orbite $\mathcal{O}$ est *transversale* si les deux conditions suivantes sont satisfaites :

$$(i)\ \forall x, y \in Out(\mathcal{O}), \forall z \in (X \setminus \mathcal{O}), (x, z) \in U \Rightarrow (y, z) \in U, \text{ et}$$

$$(ii)\ \forall x, y \in In(\mathcal{O}), \forall z \in (X \setminus \mathcal{O}), (z, x) \in U \Rightarrow (z, y) \in U.$$

Une orbite $\mathcal{O}$ est *fortement transversale* si elle est transversale et si, après la suppression des arcs $Out(\mathcal{O}) \times In(\mathcal{O})$, toutes les orbites résultantes sont fortement transversales. □

Exemple 8.3.1 Considérons le graphe (hamac) de la figure 8.1(b). Il a 7 orbites, à savoir : $\mathcal{O}_1 = \{1, 2, 3\}$, $\mathcal{O}_2 = \{1, 2\}$, $\mathcal{O}_3 = \{2, 3\}$, $\mathcal{O}_4 = \{1, 3\}$, $\mathcal{O}_5 = \{1\}$, $\mathcal{O}_6 = \{2\}$ et $\mathcal{O}_7 = \{3\}$. L'orbite $\mathcal{O}_1$ est une orbite maximale. Les orbites $\mathcal{O}_2$ et $\mathcal{O}_3$ ne sont pas maximales. Nous avons $In(\mathcal{O}_1) = \{1\}$ et $Out(\mathcal{O}_1) = \{1, 2, 3\}$. Considérons maintenant l'orbite $\mathcal{O}_3$. Elle est stable car tous les arcs $Out(\mathcal{O}_3) \times In(\mathcal{O}_3)$ sont dans $\mathcal{O}_3$. Elle est transversale car les arcs $(1, 2)$, $(1, 3)$, $(2, 4)$ et $(3, 4)$ existent dans le graphe. En fait, $\mathcal{O}_1$, $\mathcal{O}_2$ et $\mathcal{O}_3$ sont fortement stables et fortement transversales. □

8.3.2 Transformation d'un graphe de Glushkov en une expression régulière

La première étape de la transformation d'un graphe de Glushkov en une expression régulière consiste à déterminer le graphe sans orbites. Soit $G = (X, U)$ un graphe, G est appelé un *graphe sans orbites* si, pour tout x et pour tout y dans X, il n'existe aucun chemin non trivial de x à y et de y à x.

Étant donné un graphe G dont toutes les orbites sont fortement stables, nous construisons un *graphe sans orbites* en supprimant, récursivement, les arcs (x, y) qui appartiennent aux orbites maximales et tels que $x \in Out(\mathcal{O})$ et $y \in In(\mathcal{O})$. Le processus termine lorsqu'il n'y a plus d'orbites dans G. Le graphe sans orbites construit à partir de G est noté G_{wo} . Notez que le graphe sans orbites de G est défini de façon unique [CZ00].

Pendant le processus de construction de G_{wo} à partir de G, nous construisons aussi une structure hiérarchique (un arbre), appelé *hiérarchie des orbites* $\mathcal{H}$ dont chaque nœud stocke l'une des orbites maximales de G, ses portes d'entrée et de sortie. Les nœuds ont la forme $< \mathcal{O}, In(\mathcal{O}), Out(\mathcal{O}) >$. La hiérarchie des orbites est organisée en suivant la relation d'inclusion entre ensembles (inclusion des orbites), *i.e.*, pour chaque orbite $\mathcal{O}$, chaque fils de $\mathcal{O}$ est une orbite $\mathcal{O}'$ telle que $\mathcal{O}' \subset \mathcal{O}$. Dans $\mathcal{H}$, le nœud racine, notée $\mathcal{O}_r$, représente tous les nœuds de G (*i.e.*, X) et par définition $In(\mathcal{O}_r) = \emptyset$ et $Out(\mathcal{O}_r) = \emptyset$, *i.e.*, la racine de $\mathcal{H}$ est $< X, \emptyset, \emptyset >$. Par abus de notation, nous allons appeler orbite le nœud racine $\mathcal{O}_r$ de $\mathcal{H}$.

La définition suivante est utilisée pour obtenir une expression régulière à partir d'un graphe sans orbite (construit à partir d'un graphe de Glushkov donné). Chaque nœud du graphe sans orbite correspond à une expression régulière (qui, au départ, est juste la position qui identifie le nœud). À la fin du processus de réduction, le graphe n'a qu'un nœud et le contenu de ce nœud correspond à l'expression régulière représentée par le graphe.

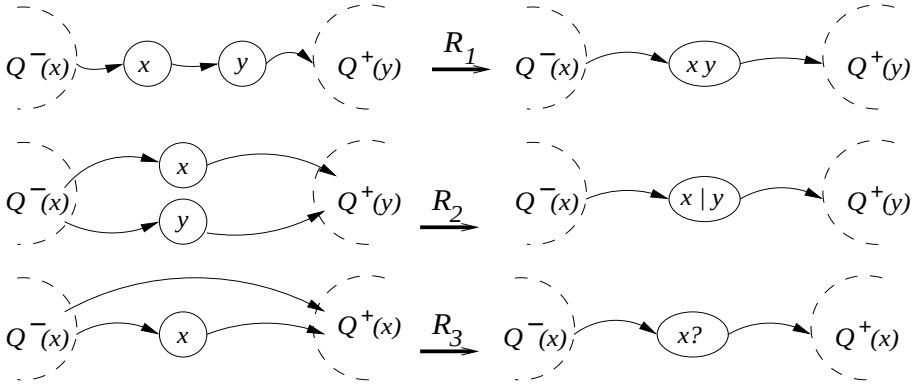


FIG. 8.2 – Règles $\mathbf{R}_1$, $\mathbf{R}_2$ et $\mathbf{R}_3$.

Définition 8.3.6 - Graphe réductible [CZ00] : Soit $G = (X, U)$ un graphe, G_{wo} est dit *réductible* s'il est possible de le réduire à un nœud en appliquant successivement l'une des trois règles $\mathbf{R}_1$, $\mathbf{R}_2$ et $\mathbf{R}_3$ décrites ci-dessous et illustrées par la figure 8.2.

Soit x un nœud dans $G_{wo} = (X, U)$. Alors $Q^-_{G_{wo}}(x) = \{y \in X \mid (y, x) \in U\}$ dénote l'ensemble de prédécesseurs immédiats de x et $Q^+_{G_{wo}}(x) = \{y \in X \mid (x, y) \in U\}$ l'ensemble de successeurs immédiats de x. Nous écrivons $Q^-(x)$ et $Q^+(x)$ lorsqu'il n'y a pas d'ambiguïté.

Règle $\mathbf{R}_1$: S'il existe deux nœuds x et y tels que $Q^-(y) = \{x\}$ et $Q^+(x) = \{y\}$, *i.e.*, le nœud x est le seul prédécesseur du nœud y et le nœud y est le seul successeur du nœud x, alors les expressions régulières associées aux nœuds x et y sont concaténées (voir figure 8.2), la nouvelle expression est associée au nœud x et le nœud y est supprimé.

Règle $\mathbf{R}_2$: S'il existe deux nœuds x et y tels que $Q^-(x) = Q^-(y)$ et $Q^+(x) = Q^+(y)$, *i.e.*, les nœuds x et y ont les mêmes successeurs et les mêmes prédécesseurs, alors on construit une expression régulière qui correspond à l'union des expressions associées aux nœuds x et y (voir figure 8.2), cette expression est associée au nœud x et le nœud y est supprimé.

Règle $\mathbf{R}_3$: S'il existe un nœud x tel que $y \in Q^-(x) \Rightarrow Q^+(x) \subset Q^+(y)$, *i.e.*, le nœud x est optionnel (voir figure 8.2), alors on supprime les arcs de $Q^-(x)$ à $Q^+(x)$ et on construit une expression régulière comme suit : si l'expression régulière d'origine, associée au nœud x, est de la forme E^+ (respectivement E) alors la nouvelle expression est E^* (respectivement $E?$). □

Notez que les règles ci-dessus ne prennent pas en considération la construction de l'expression E^+. Ce type d'expression est construit en considérant les orbites existantes dans le graphe d'origine.

Comme expliqué brièvement dans le chapitre 7, Caron et Ziadi [CZ00] proposent d'ajouter à la fin de l'expression régulière le symbole de fin # et ainsi, ils utilisent les propriétés du graphe associé à l'automate pour caractériser un automate de Glushkov. Par exemple, l'expression régulière $ab^*\#$ dénote le langage formé par les mots commençant par un a suivi d'un nombre arbitraire de b. La caractérisation des automates de Glushkov est donnée par le théorème ci-dessous.

Théorème 8.3.1 [CZ00] *$G = (X, U)$ est un graphe de Glushkov si et seulement si les trois conditions suivantes sont respectées :*

1. *G est un hammac;*
2. *Chaque orbite maximale dans G est fortement stable et fortement transversale; et*
3. *G_{wo} est réductible.*

Selon le lemme ci-dessous nous pouvons associer une orbite dans un graphe de Glushkov G à

une fermeture (étoile de Kleene) dans l'expression régulière correspondante. En d'autres termes, les nœuds dans la hiérarchie des orbites de G (sauf le nœud racine) représentent les positions de sous-expressions étoilées de l'expression régulière utilisée dans la construction de l'automate de Glushkov. Cette propriété à été prouvée dans [CF03, BW92] et elle est très importante pour notre méthode, comme nous verrons plus tard, dans le processus de construction et de classification des expressions régulières candidates.

Lemme 8.3.1 [CZ00] *Soit $G = (X, U)$ un graphe qui respecte les propriétés 2 et 3 du théorème 8.3.1. Soit $\mathcal{O}$ une orbite maximale dans G. En appliquant les règles $\boldsymbol{R}_1$, $\boldsymbol{R}_2$ et $\boldsymbol{R}_3$ dans G_{wo}, l'orbite sera réduite à un seul nœud, en considérant que les règles $\boldsymbol{R}_1$ et $\boldsymbol{R}_2$ sont appliquées à toute paire (x, y) de $(\mathcal{O} \times \mathcal{O})$ ou de $[(X \setminus \mathcal{O}) \times (X \setminus \mathcal{O})]$.*

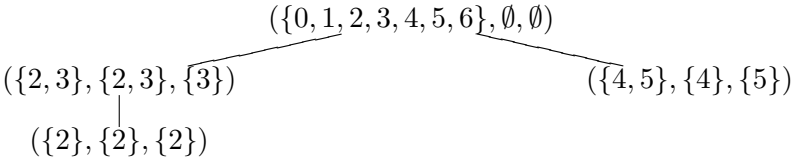
Il est important de remarquer que le processus de réduction travaille de l'intérieur vers l'extérieur des orbites en respectant $\mathcal{H}$. La réduction commence au niveau le plus bas de la hiérarchie (des orbites les plus internes aux orbites les plus englobantes). La hiérarchie $\mathcal{H}$ guide l'application des règles de réduction. Ci-dessous, nous présentons un algorithme qui construit la paire $(G_{wo}, \mathcal{H})$ à partir d'un graphe de Glushkov $G = (X, U)$. L'algorithme est basé sur la définition des portes d'entrées et de sortie (définition 8.3.3) et la définition de la transformation d'un graphe de Glushkov en un graphe de Glushkov sans orbites.

Algorithme 8.3.1 - *Construction de la paire* $(G_{wo}, \mathcal{H})$

01. `Entrée: un graphe de Glushkov` $G = (X, U)$
02. `Sortie: la hiérarchie des orbites` $\mathcal{H}$ `de` G `et` G_{wo}.
03. $G' = (X', U')$, $X' = X$ `and` $U' = U$
04. `Let the root of` $\mathcal{H}$ `be` $(X, \emptyset, \emptyset)$
05. `//` la fonction *GetStronglyConnectedComp* retourne des ensembles
06. `//` de nœuds $\mathcal{O}$ qui appartiennent aux composantes fortement connexes ayant
07. `//` au moins un arc dans G'.
08. `while` $((SCC = GetStronglyConnectedComp(G')) != null)$ `do {`
09. `for each` $\mathcal{O}$ `in` SCC `do {`
10. $In_{\mathcal{O}} = InputGate(\mathcal{O}, G')$
11. $Out_{\mathcal{O}} = OutGate(\mathcal{O}, G')$
12. `insert` $(\mathcal{O}, In_{\mathcal{O}}, Out_{\mathcal{O}})$ `into` $\mathcal{H}$ `as child of a node` $\mathcal{O}'$ `such that` $\mathcal{O} \subset \mathcal{O}'$
13. `and` $(\forall \mathcal{O}'' \in \mathcal{H}, \mathcal{O} \subset \mathcal{O}'' \Rightarrow \mathcal{O}' \subset \mathcal{O}'')$
14. `for each` (x, y) `in` U' `and` $x \in Out_{\mathcal{O}}$ `and` $y \in In_{\mathcal{O}}$ `do {`
15. `delete` (x, y) `from` U' `}`
16. `}`
17. $G_{wo} = G'$ `}`
18. `return` G_{wo} `and` $\mathcal{H}$

La hiérarchie des orbites $\mathcal{H}$ a une structure d'arbre. Nous la représentons par un n-uplet d'ensemble de positions, qui correspond au parcours des nœuds de l'arbre en postordre.

Par exemple, étant donné le graphe de Glushkov de la figure 8.3 qui représente l'automate d'états finis de l'expression régulière $E = a(b^*c)^*(de)^+f$ ($\overline{E} = 1(2^*3)^*(4\ 5)^+6$), la hiérarchie des orbites $\mathcal{H}$ est:



et sa représentation sous la forme d'un n-uplet est ⟨ ({2}, {2}, {2}), ({2, 3}, {2, 3}, {3}), ({4, 5}, {4}, {5}), ({0, 1, 2, 3, 4, 5, 6}, ∅, ∅) ⟩. Pour simplifier l'écriture, dorénavant, on notera $\mathcal{H}$ sans les portes d'entrées et de sorties.

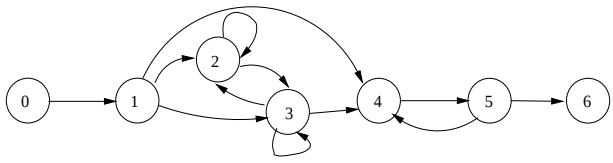


FIG. 8.3 – Un graphe de Glushkov construit à partir de l'automate d'états finis de l'expression régulière $E = a(b^*c)^*(de)^+f$.

L'exemple ci-dessous illustre certaines étapes du processus de réduction d'un graphe de Glushkov.

Exemple 8.3.2 Considérons le graphe G de la figure 8.1(b). Le graphe sans orbite correspondant est montré dans la figure 8.4(a). De l'exemple 8.3.1, nous remarquons que le graphe G a sept orbites, cependant, seulement les orbites $\mathcal{O}_3$ et $\mathcal{O}_1$ participent de la hiérarchie des orbites de G. En effet, la hiérarchie des orbites est $\mathcal{H}$=$\langle\{2,3\}, \{1,2,3\}, \{0,1,2,3,4\}\rangle$.

Dans la figure 8.4 on voit des étapes du processus de réduction. La figure 8.4(b) est le résultat de l'application de la règle $\mathbf{R}_2$. Notez que le nœud (2|3) représente l'orbite $\mathcal{O}_3$. Dans ce cas, nous construisons la fermeture positive $(2|3)^+$ (figure 8.4(c)). Après l'application de la règle $\mathbf{R}_3$, l'expression $(2|3)^+$ devient optionnelle (figure 8.4(d)). En appliquant la règle $\mathbf{R}_1$ et en prenant en considération l'orbite $\mathcal{O}_1$, on obtient le graphe de la figure 8.4(e). La figure 8.4(f) est le résultat de l'application de la règle $\mathbf{R}_3$. Si le processus de réduction continue, on applique deux fois la règle $\mathbf{R}_1$ et on obtient $0(1(2|3)^*)^*4$ qui est facilement traduite en $(a(b|c)^*)^*d$ en utilisant un morphisme (où 0 est l'état initial et chaque position correspond au symbole dans l'expression régulière d'origine). □

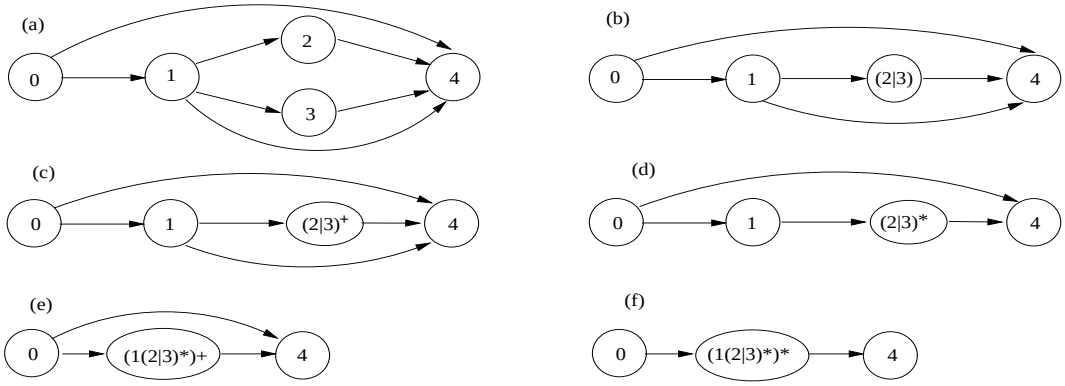


FIG. 8.4 – Un exemple de réduction.

Notons que les orbites maximales présentées dans l'exemple ci-dessus correspondent aux sous-expressions étoilées de l'expression $(a(b|c)^*)^*d$. Les positions de $\mathcal{O}_1$ (*i.e.*, 1, 2 et 3) représentent $(a(b|c)^*)^*$ et les positions de $\mathcal{O}_3$ (*i.e.*, 2 et 3) représentent $(b|c)^*$.

8.4 Évolution de schéma XML

Dans cette section, nous présentons une méthode pour assister l'administrateur dans l'évolution des schémas XML. Nous rappelons que le schéma est défini par un automate d'arbre ascendant déterministe $\mathcal{A}$ construit à partir d'une DTD (non ambiguë).

Avant de présenter la méthode d'évolution de schémas, nous proposons une notion très simple de distance entre deux expressions régulières fondée sur le nombre de positions des expressions

régulières indicées correspondantes. Cette distance est utile pour caractériser les nouvelles expressions régulières construites à partir des expressions régulières qui définissent le modèle de contenu des éléments. De plus, cette notion nous permet de montrer que notre méthode n'insère pas de symboles de plus dans les expressions régulières candidates.

Définition 8.4.1 - Distance entre deux expressions régulières: Soit E et E' des expressions régulières. La *distance* entre E et E', notée par $\mathcal{D}(E, E')$, est $\mathcal{D}(E, E') = |card(Pos(E)) - card(Pos(E'))|$; où $card(Pos(E)$ représente le nombre d'éléments de l'ensemble de positions de E. □

Exemple 8.4.1 Soient $E_1 = a(b\ c)*$ et $E_2 = a(b\ c\ d?)^*$ deux expressions régulières. Soient $Pos(E_1) = \{1, 2, 3\}$ et $Pos(E_2) = \{1, 2, 3, 4\}$ les ensembles de positions de E_1 et E_2, respectivement. La distance entre E_1 et E_2 est calculée comme suit : la cardinalité de $Pos(E_1)$ est égal à 3 et la cardinalité de $Pos(E_2)$ est égal à 4 et ainsi, $\mathcal{D}(E_1, E_2) = |3 - 4| = | - 1| = 1$. □

Les mises à jour sont vues comme des modifications exécutées sur un arbre qui représente un document XML. Les mises à jour invalides produisent des arbres XML qui ne peuvent plus être reconnus par l'automate $\mathcal{A}$. Les mises à jour peuvent être de trois types :
(1) L'insertion ou la suppressions de nœuds du type élément (ou data),
(2) l'insertion ou la suppression de nœuds du type attribut et
(3) le renommage des étiquettes (attributs ou éléments).

Les expressions régulières candidates construites à partir des mises à jour invalides du type (2) sont définies de façon immédiate : soit a, $<S_{compulsory}, S_{optional}>$, $E \rightarrow q_a$ une règle de transition. Si un attribut obligatoire est supprimé alors le nouveau schéma est obtenu en déplaçant l'état qui représente l'attribut supprimé vers l'ensemble $S_{optional}$. Si un nouvel attribut est inséré, alors le nouveau schéma est obtenu en insérant cet attribut (l'état qu'il représente) comme optionnel. Il est important de noter que, dans notre approche, aucun attribut ne peut être inséré comme obligatoire car la méthode d'évolution proposée préserve la cohérence des documents valides préexistant sans les modifier.

L'opération de renommage peut être vue comme une opération de suppression immédiatement suivie d'une opération d'insertion. Alors, le traitement des opérations du type (3) peut être modélisé par les cas (1) et (2) ci-dessus. Bien que les opérations soient traitées une par une par notre algorithme `GREC`, une opération du type (3) qui correspond en fait à deux opérations, peut être traitée en appelant `GREC` deux fois.

Ainsi, la méthode proposée est focalisée sur l'évolution du schéma déclenchée par des opérations de mises à jour du type (1). Dans ce cas, l'évolution revient à modifier l'expression régulière E qui détermine les sous-éléments qu'un nœud dans un arbre XML peut avoir. Le problème de mise à jour de schéma peut être formulé comme suit :

1. L'insertion (respectivement la suppression) d'un arbre t_1 comme fils d'un nœud a dans l'arbre t signifie changer le mot $w \in L(E)$ obtenu à partir de la concaténation des états des fils de a pendant l'exécution de $\mathcal{A}$. Notons que le nouveau mot w' est obtenu à partir de w avec le nouvel état inséré dans le cas d'insertion ou la suppression d'un état préexistant dans w dans le cas de suppression. Cet état est celui associé à la racine de t_1.
2. Étant donné une expression régulière E ($w \in L(E)$) et w', notre problème est de proposer des nouvelles expressions régulières E' telles que $\mathcal{D}(E, E') = 1$ pour l'insertion et $\mathcal{D}(E, E') = 0$ pour la suppression et dont les langages contiennent (au minimum) le mot d'état w' et $L(E)$.

Nous rappelons que pour valider le document, un automate de Glushkov M_E est utilisé. L'automate est construit, en utilisant la méthode proposée dans [CZ00], à partir de chaque

expression régulière E qui apparaît dans les règles de transitions de $\mathcal{A}$. Dans M_E, chaque état (hormis l'état initial) correspond à la position des symboles dans l'expression régulière indicée $\overline{E}$. Il y a toujours un unique état final, qui correspond à la position de la marque de fin de E (#).

8.4.1 Localisation des modifications

La méthode d'évolution s'appuie sur les états de l'automate M_E pour proposer des candidats qui peuvent remplacer l'expression régulière d'origine. À cet effet, trois états sont cherchés pour être utilisés pour la construction des candidats. La définition ci-dessous formalise ces trois états.

Définition 8.4.2 État le plus proche à droite (s_{nr}), état le plus proche à gauche (s_{nl}) et état spécial (s_{new}) : Soit w un mot accepté par l'automate M_E. Soient p une position dans w ($0 \leq p \leq n$) et $w[p]$ la représentation du symbole dans p. Soit $w_{ins} = w[0 : n+1]$ le mot construit à partir de w en insérant un nouveau symbole à la position p et $w_{del} = w[0 : n-1]$ le mot construit en supprimant le symbole $w[p]$ de w (p est la position de mise à jour).
L'*état le plus proche à gauche*[3] (s_{nl}) et l'*état le plus proche à droite*[4] (s_{nr}) sont définis comme suit :

- $s_{nl} = \hat{\delta}(q_0, \alpha)$, avec $\alpha = w[0 : p-1]$ ou $\alpha = w_{ins}[0 : p-1]$ ou $\alpha = w_{del}[0 : p-1]$.
- $s_{nr} = \hat{\delta}(q_0, \alpha)$, avec $\alpha = w[0 : p]$ ou $\alpha = w_{del}[0 : p]$.

L'état spécial, noté par s_{new}, est défini comme suit:
(i) un nouvel état (*i.e.*, $s_{new} \notin Q$) à ajouter dans l'automate M_E qui représente le symbole $w[p]$ dans w_{ins}, ou
(ii) un état qui doit devenir optionnel dans M_E, défini sur le mot w comme $s_{new} = \hat{\delta}(q_0, \alpha)$ avec $\alpha = w[0 : p]$, *i.e.*, l'état qui représente $w[p]$. □

Exemple 8.4.2 Soit $E = a(b\ c)^*d$ une expression régulière. Soient $\delta(0, a) = 1$, $\delta(1, b) = 2$, $\delta(1, d) = 4$, $\delta(2, c) = 3$, $\delta(3, b) = 2$ et $\delta(3, d) = 4$ les règles de transition de l'automate d'états finis M_E construit à partir de E. Étant donné le mot $w = abcbcd$ tel que $w \in L(E)$, le symbole x est inséré dans la position 2 de w, donnant $w' = abxcbcd$ tel que $w' \notin L(E)$. Les états calculés à partir de M_E et w sont :
$s_{nl} = \hat{\delta}(0, ab)$, c'est-à-dire, l'état 2 ($\alpha = w[0 : p-1]$ de la définition 8.4.2).
$s_{nr} = \hat{\delta}(0, abc)$, c'est-à-dire, l'état 3 ($\alpha = w[0 : p]$ de la définition 8.4.2).
s_{new} est un état qui n'appartient pas aux états de M_E, par exemple, 5. □

Notons que s_{nl} et s_{nr} existent toujours dans M_E car le mot w appartient au langage accepté par M_E et que si M_E est déterministe (comme préconisé par W3C), alors les états s_{nl} et s_{nr} sont uniques.

Sans perte de généralité, nous pouvons considérer qu'une requête d'insertion correspond toujours à l'insertion d'un nouveau symbole dans l'expression régulière E (même si le symbole existe déjà dans E). Ainsi, pour accepter un nouveau mot, il faut insérer un nouvel état dans M_E. Cet état (s_{new}) doit être ajouté à M_E et il faut qu'il existe une transition de s_{nl} à s_{new}. Cependant, il faut faire d'autres modifications dans M_E pour que le graphe associé à M_E soit un graphe de Glushkov (les propriétés du théorème 8.3.1 doivent être respectées). Ces modifications dépendent des caractéristiques des positions de s_{nl} et s_{nr} dans le graphe. Nous traitons les suppressions de la même façon : les propriétés des graphes de Glushkov doivent être respectées.

Le processus d'évolution des schémas est divisé en trois étapes distinctes, d'abord il faut construire la paire $(G_{wo}, \mathcal{H})$ de G et trouver les états s_{nl}, s_{nr} et s_{new}, ensuite `GREC` est appelé pour construire les expressions régulières candidates et, finalement, les candidats sont présentés

[3] The nearest left state.
[4] The nearest right state.

```
01.  function GraphToRegExp(G_wo, H) {
02.       if G_wo n'a qu'un nœud {
03.          return l'expression règulière dans le nœud de G_wo }
04.          else{
05.               R_i := ChooseRule(G_wo, H);
06.               (G_new, H_new) := ApplyRule(R_i, G_wo, H);
07.               return GraphToRegExp(G_new, H_new);
08.          }     }
```

FIG. 8.5 – L'algorithme de réduction proposé par [CZ00]

à l'utilisateur qui peut choisir l'un d'entre eux pour remplacer l'expression régulière d'origine. Il est clair que si le processus qui cherche la paire (s_{nl}, s_{nr}) trouve n paires (le cas des automates non déterministes), `GREC` est appelé n fois. Cependant, dans ce travail, le processus qui cherche la paire (s_{nl}, s_{nr}) trouve toujours une seule paire (du fait que les automates sont déterministes).

8.4.2 L'algorithme `GREC`

L'algorithme `GREC` (**G**enerate **R**egular **E**xpression **C**hoices) construit les expressions règulières candidates, à partir de la méthode de réduction proposée par [CZ00] (figure 8.5).

La figure 8.6 présente l'algorithme `GREC`. Cet algorithme prend cinq paramètres en entrée : un graphe sans orbites G_{wo}, la hiérarchie des orbites $\mathcal{H}$, les deux nœuds qui correspondent aux nœuds s_{nl} et s_{nr}, et le nœud s_{new}. Le processus de réduction proposé par [CZ00] est représenté par les procédures `ChooseRule`, `ApplyRule` et `GraphToRegExp`.

`ChooseRule` utilise les informations sur les orbites pour choisir la règle à appliquer, elle retourne la règle à appliquer et les nœuds affectés par la règle (dans la structure R_i). Les règles sont cherchées selon les conditions imposées par la définition 8.3.6. La fonction `ApplyRule` transforme le graphe en entrée en un autre graphe selon la règle utilisée et met à jour la hiérarchie des orbites. Enfin, la fonction `GraphToRegExp` transforme récursivement un graphe en une expression régulière en appelant les fonctions `ChooseRule` et `ApplyRule` avec le nouveau graphe et la nouvelle hiérarchie des orbites.

L'algorithme `GREC` étend l'algorithme `GraphToRegExp` comme suit : à chaque pas du processus de réduction, il vérifie si la règle choisie affecte l'un des nœuds s_{nr} ou s_{nl} et, si c'est le cas, il modifie le graphe d'origine :
(i) soit en insérant le nœud s_{new} (insertions)
(ii) soit en rendant le nœud s_{new} optionnel (suppressions).
Cette vérification et cette modification sont assurées par la fonction `LookForGraphAlternative`.

Chaque pas du processus de réduction amène à remplacer des nœuds du graphe par un nœud associé à une expression régulière plus complexe. La réduction termine lorsque le graphe n'est plus formé que par un seul nœud, ce nœud contenant alors l'expression régulière complète. Notre but est d'insérer un nouveau nœud dans le graphe et de proposer différentes solutions à partir de ce nouveau graphe. C'est dans les lignes 07 et 08 de la figure 8.6 que les différentes solutions sont calculées.

```
   function GREC(G_wo, H, s_nl,s_nr, s_new) {
01.  setRegExp = ∅ ;
02.  if G_wo n'a qu'un nœud {
03.     return ∅ }
04.     else{
05.       R_i := ChooseRule(G_wo, H);
06.       // les lignes suivante étendent les lignes de l'algorithme d'origine GraphToRegExp
07.       for each (G_new, H_new):= LookForGraphAlternative(G_wo,H,R_i,s_nl,s_nr,s_new) do
08.            setRegExp = setRegExp ∪{ GraphToRegExp(G_new, H_new) };
09.       // fin de l'extension
10.       (G'_wo,H') := ApplyRule(R_i, G_wo, H);
11        setRegExp = setRegExp ∪ GREC(G'_wo, H', s_nl, s_nr, s_new);
12.     }
   return setRegExp }
```

FIG. 8.6 – L'algorithme pour calculer les expressions régulières candidates.

Étant donné un graphe de Glushkov sans orbite G_{wo} et sa hiérarchie d'orbites $\mathcal{H}$, la vérification de la possibilité d'insertion du nouveau nœud à chaque pas du processus de réduction du graphe (de l'automate) d'origine est faite comme suit :

1. On applique `GREC` sur G_{wo}, $\mathcal{H}$ et les nœuds s_{nl} et s_{nr} : avant d'appliquer une des règles de réduction sur G_{wo}, on exécute des tests, qui sont différents pour chaque règle $\mathbf{R}_1$, $\mathbf{R}_2$ ou $\mathbf{R}_3$ (ligne 05).
2. Avant l'application d'une règle R_i dans le processus de réduction, on teste dans la fonction `LookForGraphAlternative` si les nœuds s_{nr} ou s_{nl} sont affectés par R_i.
   (a) Si s_{nl} ou s_{nr} sont affectés par R_i alors le nœud s_{new} est inséré dans le graphe G_{wo}. Cette opération génère un ou plusieurs graphes G_{new} (ligne 07) auxquels on applique la méthode de [CZ00] (ligne 08).
   (b) Sinon, aucune modification n'est exécutée sur G_{wo}.
3. L'exécution de `GREC` sur G_{wo} continue (ligne 10) : on applique R_i sur G_{wo} et le processus continue (ligne 11) comme expliqué dans les items 1 et 2. Le processus s'arrête lorsque G_{wo} a seulement un nœud (ligne 02).
4. L'ordre d'application des règles $\mathbf{R}_1$, $\mathbf{R}_2$ et $\mathbf{R}_3$ ainsi que les nœuds sur lesquels les appliquer sont donnés par $\mathcal{H}$.

Notons que les pas représentés par les items $2(a)$ et $2(b)$ ci-dessus sont exécutés par la fonction `LookForGrapheAlternative` qui est la responsable de la construction des nouveaux graphes candidats qui deviendront les expressions régulières candidates.

La fonction `LookForGrapheAlternative` : génération des graphes candidats

On remarque que le rôle de la fonction `LookForGrapheAlternative` est essentiel puisque c'est dans cette procédure que sont générées les solutions proposées par `GREC`. Les principes de génération de ces solutions sont maintenant présentés.

Les modifications apportées au graphe à réduire sont guidées par les règles $\mathbf{R}_1$, $\mathbf{R}_2$ et $\mathbf{R}_3$ et par les informations sur les orbites du graphe d'origine. En effet, la procédure `LookForGraphAlternative` utilise les cas présentés dans les définitions 8.4.3 à 8.4.6 ci-dessous (illustrées dans les tables des figures 8.7 à 8.10).

Définition 8.4.3 Candidats générés à partir de $\mathbf{R}_1$: Soit $G_{wo} = (X, U)$ un graphe (sans orbite) de Glushkov et $x, y \in X$ deux nœuds sur lesquels $\mathbf{R}_1$ peut être appliquée. Soit s_{new} l'état à être inséré dans G_{wo}. Le symbole a dans w' qui correspond à s_{new} est différent de celui correspondant à l'état x. Les graphes $G^i = (X^i, U^i)$ qui peuvent être proposés à partir de G_1 sont définis comme suit :

Cas 1 $x = s_{nl}$ et $y = s_{nr}$:
$\mathbf{G}^1$: $X^1 = X \cup \{s_{new}\}$; $U^1 = U \cup \{(x, s_{new})\} \cup \{(s_{new}, y)\}$.
Cas 2 $x = s_{nr}$ et $y = s_{nl}$:
$\mathbf{G}^1$: $X^1 = X \cup \{s_{new}\}$; $U^1 = U \cup \{(y, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(y)\}$.
$\mathbf{G}^2$: $X^2 = X \cup \{s_{new}\}$; $U^2 = U \cup \{(s_{new}, x)\} \cup \{(z, s_{new}) \mid z \in Q^-(x)\}$. □

La définition 8.4.3 ainsi que la figure 8.7 montrent comment la fonction `LookForGraphAlternative` construit les nouveaux graphes lorsque la règle $\mathbf{R}_1$ est appliquée. Les conditions pour le premier cas sont : (i) le nœud x correspond au nœud s_{nl} et (ii) le nœud y correspond au nœud s_{nr}. Dans ce cas, un nouveau graphe G^1 est construit avec le nœud s_{new} inséré comme un nœud optionnel entre x et y.

Par exemple, supposons que $E = ab$ et $w = ab$. Si $w' = anb$, les expressions régulières résultantes du nouveau graphe seront $an?b$ et an^*b. Dorénavant, nous emploierons la notation $E!$ pour représenter les expressions $E?$ et E^*, ainsi $an!b$ représentera $an?b$ et an^*b.

Le deuxième cas de cette définition correspond au cas où s_{nl} est une porte de sortie d'une orbite $\mathcal{O}$ et s_{nr} est une porte d'entrée de $\mathcal{O}$. Deux nouveaux graphes sont alors construits à partir du graphe d'origine (G^1 et G^2).

Par exemple, si $E = a(bc)^*d$, $w = abcbcd$ et $w' = abcnbcd$, les expressions régulières correspondant à G^1 et G^2 sont $E = a(bcn!)^*d$ et $E = a(n!bc)^*d$, respectivement.

Remarquer que dans le cadre d'application de la règle $\mathbf{R}_1$, il existe des situations dans la construction d'une expression régulière candidate où il n'est pas nécessaire d'insérer le nouveau nœud. La définition 8.4.3 teste si s_{new} représente le même symbole que x dans l'expression régulière à construire. Si le symbole représenté par s_{new} est égal au symbole représenté par x lors du test de la définition 8.4.3, s_{new} n'est pas inséré dans le graphe. Néanmoins, on ajoute une nouvelle orbite $(\{x\}, \{x\}, \{x\})$ dans la hiérarchie des orbites.

Par exemple, étant donné $E = abc$, $w = abc$ et $w' = abbc$, la définition 8.4.3 construit les candidates $abb!c$ et ab^+c car s_{new} représente b et x, lors de l'application de la définition 8.4.3, représente aussi b (et y représente c).

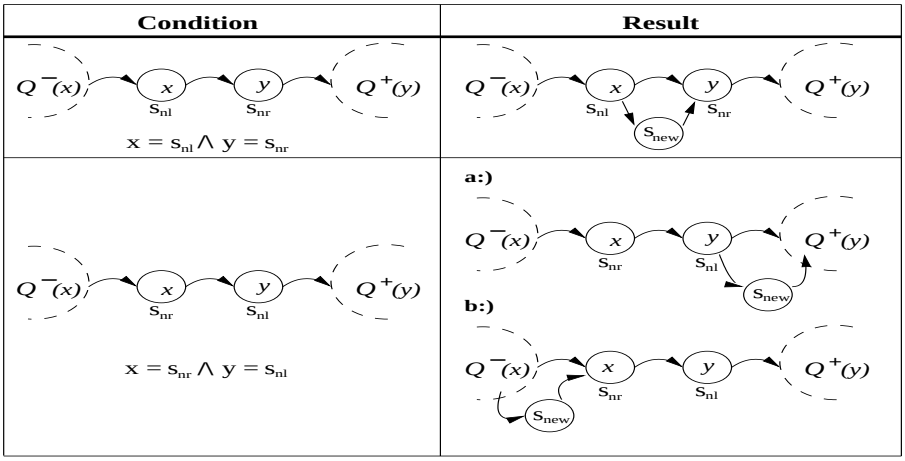


FIG. 8.7 – Conditions et modifications à tester avant d'appliquer la règle $\mathbf{R}_1$.

Définition 8.4.4 Candidats générés à partir de $\mathbf{R}_2$: Soit $G_{wo} = (X, U)$ un graphe de Glushkov et $x, y \in X$ deux nœuds sur lesquels $\mathbf{R}_2$ peut être appliquée. Les graphes $G^i = (X^i, U^i)$ construits à partir de G_1 sont définis comme suit :

Cas 1 $x = s_{nl}$ et $s_{nr} \in Q^+(x)$:
$\mathbf{G}^1 : X^1 = X \cup \{s_{new}\}$; $U^1 = U \cup \{(x, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(x)\}$.
Cas 2 $x = s_{nr}$ et $s_{nl} \in Q^-(x)$:
$\mathbf{G}^1$: $X^1 = X \cup \{s_{new}\}$; $U^1 = U \cup \{(s_{new}, x)\} \cup \{(z, s_{new}) \mid z \in Q^-(x)\}$.
Cas 3 $s_{nl} \in Q^-(x)$ et $s_{nr} \in Q^+(x)$:
$\mathbf{G}^1$: $X^1 = X \cup \{s_{new}\}$; $U^1 = U \cup \{(z, s_{new}) \mid z \in Q^-(x)\} \cup \{(s_{new}, v) \mid v \in Q^+(x)\}$.
Cas 4 ($s_{nl} = x$ et $s_{nr} = y$) ou ($s_{nl} = y$ et $s_{nr} = x$) :
$\mathbf{G}^1$: $X^1 = X \cup \{s_{new}\}$; $U^1 = U \cup \{(z, s_{new}) \mid z \in Q^-(x)\} \cup \{(s_{new}, v) \mid v \in Q^+(x)\}$.
$\mathbf{G}^2$: $X^2 = X \cup \{s_{new}\}$; $U^2 = U \cup \{(x, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(x)\}$.
$\mathbf{G}^3$: $X^3 = X \cup \{s_{new}\}$; $U^3 = U \cup \{(s_{new}, y)\} \cup \{(z, s_{new}) \mid z \in Q^-(y)\}$. □

La définition 8.4.4 (illustrée par la figure 8.8) traite des conditions à tester et des modifications à faire dans le graphe lorsque la règle $\mathbf{R}_2$ est appliquée. Dans le premier cas $x = s_{nl}$ et $s_{nr} \in Q^+(x)$ alors s_{new} est inséré comme successeur de s_{nl}.

Par exemple, si $E = a(b|c)d$, $w = abd$ et $w' = abnd$, alors l'expression régulière construite à partir du graphe modifié est $a(bn!|c)d$.

Le deuxième cas est symétrique. Le troisième cas construit un graphe dans lequel le nœud s_{new} est placé en disjonction par rapport à x et à y.

Par exemple, si $E = a(b?|c)d$, $w = ad$ et $w' = and$, alors l'expression régulière obtenue du nouveau graphe (G^1) est $a(b?|c|n!)d$.

Le quatrième cas construit des graphes comme les cas 1, 2 et 3, cependant, dans ce cas, les nœuds s_{nl} et s_{nr} représentent x et y.

Par exemple, si $E = a(b|c)^*d$, $w = abcd$ et $w' = abncd$, alors les expressions régulières obtenues à partir des nouveaux graphes sont $a(bn!|c)^*d$, $a(b|n!c)^*d$ et $a(b|c|n!)^*d$, résultantes des graphes G^1, G^2 et G^3, respectivement.

Définition 8.4.5 Candidats générés à partir de $\mathbf{R}_3$: Soit $G_{wo} = (X, U)$ un graphe de Glushkov et $x \in X$ un nœud sur lequel $\mathbf{R}_3$ peut être appliquée. Les graphes $G^i = (X^i, U^i)$ construits à partir de G_1 sont définis comme suit :

Cas 1 ($s_{nl} \in Q^-(x)$ et $s_{nr} \in Q^+(x)$) ou ($s_{nl} \in Q^-(x)$ et $s_{nr} \notin Q^+(x)$ et $s_{nr} \notin \{x\}$) ou
($s_{nl} \notin Q^-(x)$ et $s_{nr} \in Q^+(x)$ et $s_{nl} \notin \{x\}$):
$\mathbf{G}^1$: $X^1 = X \cup \{s_{new}\}$; $U^1 = U \cup \{(s_{new}, x)\} \cup \{(z, s_{new}) \mid z \in Q^-(x)\}$.
$\mathbf{G}^2$: $X^2 = X \cup \{s_{new}\}$; $U^2 = U \cup \{(x, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(x)\}$.
$\mathbf{G}^3$: $X^3 = X \cup \{s_{new}\}$; $U^3 = U \cup \{(z, s_{new}) \mid z \in Q^-(x)\} \cup \{(s_{new}, v) \mid v \in Q^+(x)\}$. □

La définition 8.4.5 et la figure 8.9 présentent les conditions à tester et les modifications à faire si la règle $\mathbf{R}_3$ est appliquée. Les conditions sont : (i) $s_{nl} \in Q^-(x)$ et $s_{nr} \in Q^+(x)$ ou (ii) $s_{nl} \in Q^-(x)$ et $s_{nr} \notin Q^+(x)$ ou (iii) $s_{nl} \notin Q^-(x)$ et $s_{nr} \in Q^+(x)$. Les conditions (ii) et (iii) traitent le cas similaire au deuxième cas de la définition 8.4.3. L'algorithme propose trois solutions possibles, à savoir, insérer s_{new} avant, après et en disjonction avec le nœud x.

Par exemple, étant donné $E = ab?c$, $w = ac$ et $w' = anc$, les expressions régulières obtenues à partir des graphes modifiés sont : $an!b?c$, $ab?n!c$ et $a(n! \mid b?)c$.

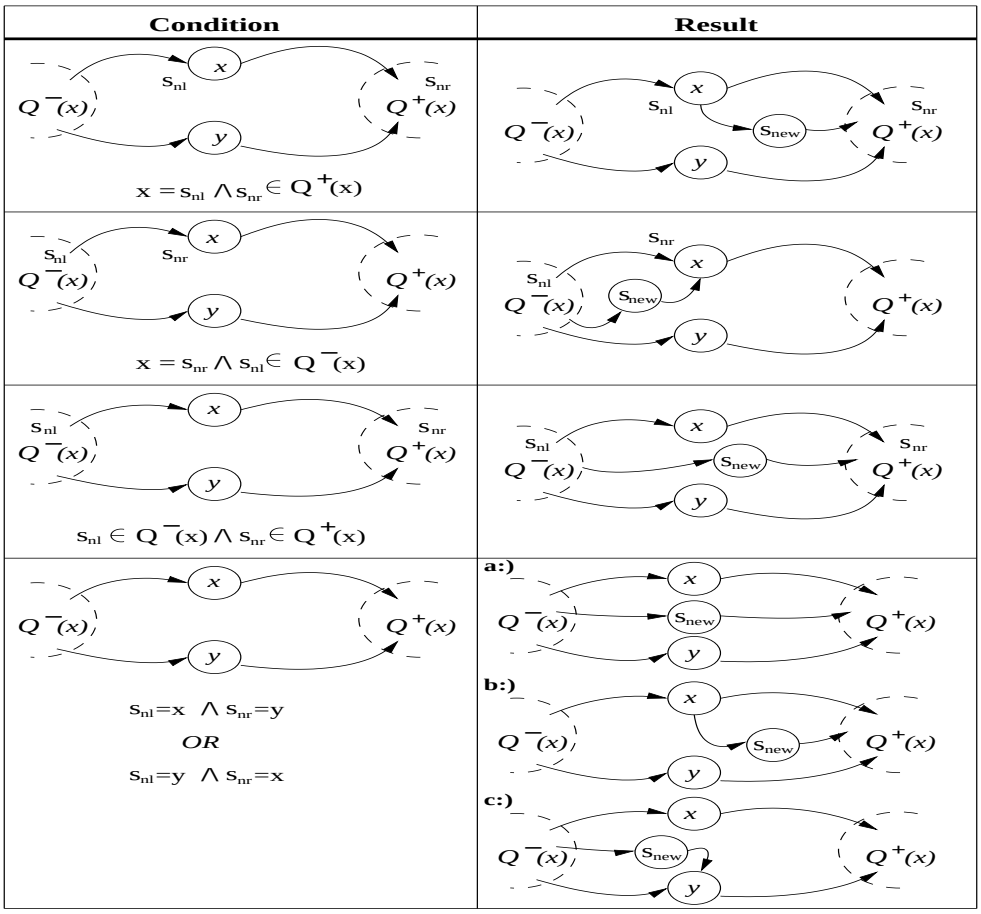


FIG. 8.8 – Conditions et modifications à tester avant d'appliquer la règle $\mathbf{R}_2$.

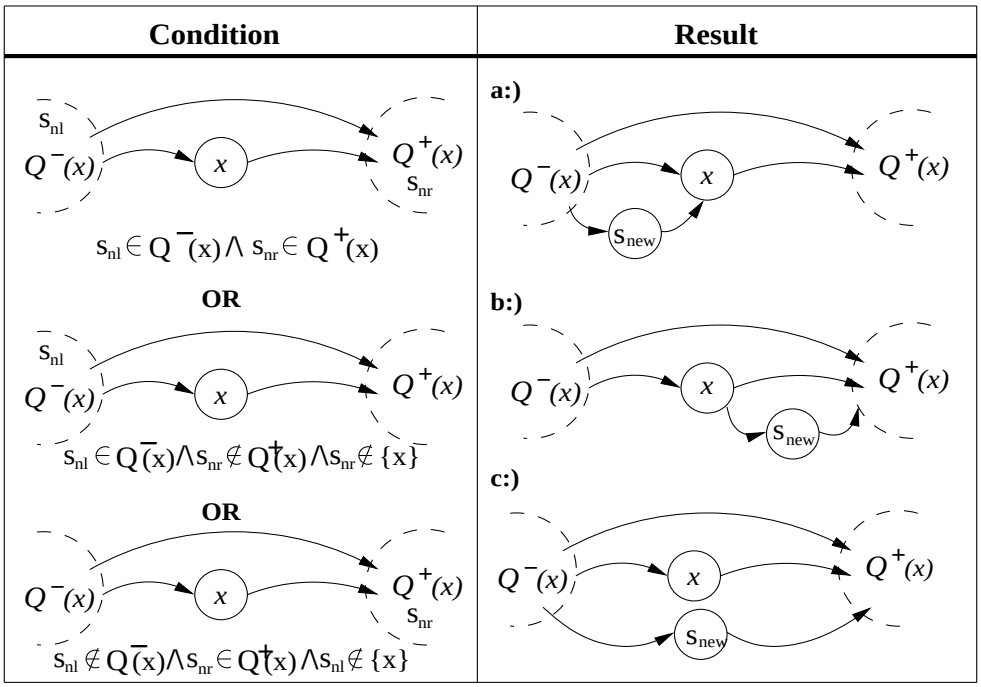


FIG. 8.9 – Conditions et modifications à tester avant d'appliquer la règle $\mathbf{R}_3$.

Il est important de remarquer que les règles $\mathbf{R}_1$, $\mathbf{R}_2$ et $\mathbf{R}_3$ sont d'abord appliquées à l'intérieur de chaque orbite du graphe, en respectant la hiérarchie des orbites. Par ailleurs, pendant le processus de réduction, chaque orbite $\mathcal{O}$ du graphe d'origine est réduite à un nœud, qui contient l'expression qui sera décorée avec un plus ($^+$). Avant d'appliquer la décoration, on doit considérer l'insertion du nœud s_{new} dans l'orbite $\mathcal{O}$. La définition 8.4.6 résume les conditions selon lesquelles le graphe d'origine peut être changé à cette étape de la réduction.

Définition 8.4.6 Candidats générés à partir de la réduction complète d'une orbite : Soit $G_{wo} = (X, U)$ un graphe de Glushkov. Soit $\mathcal{O}$ une orbite réduite à un nœud $z \in X_1$. Les graphes $G^i = (X^i, U^i)$ construits à partir de G_1 sont définis comme suit :

Cas 1 $s_{nr} \in In(\mathcal{O})$ et $s_{nl} \in Out(\mathcal{O})$:
- $\mathbf{G}^1$: $X^1 = X \cup \{s_{new}\}$; $U^1 = U \cup \{(s_{new}, z)\} \cup \{(x, s_{new}) \mid x \in Q^-(z)\}$.
- $\mathbf{G}^2$: $X^2 = X \cup \{s_{new}\}$; $U^2 = U \cup \{(z, s_{new})\} \cup \{(s_{new}, x) \mid x \in Q^+(z)\}$.
- $\mathbf{G}^3$: $X^3 = X \cup \{s_{new}\}$; $U^3 = U \cup \{(x, s_{new}) \mid x \in Q^-(z)\} \cup \{(s_{new}, x) \mid x \in Q^+(z)\}$.

Cas 2 $s_{nl} \in Q^-(z)$ et $s_{nr} \in \mathcal{O}$:
- $\mathbf{G}^1$: $X^1 = X \cup \{s_{new}\}$; $U^1 = U \cup \{(s_{new}, z)\} \cup \{(x, s_{new}) \mid x \in Q^-(z)\}$.
- $\mathbf{G}^2$: $X^2 = X \cup \{s_{new}\}$; $U^2 = U \cup \{(v, s_{new}) \mid v \in Q^-(z)\} \cup \{(s_{new}, x) \mid x \in Q^+(z)\}$.

Cas 3 $s_{nl} \in \mathcal{O}$ et $s_{nr} \in Q^+(z)$:
- $\mathbf{G}^1$: $X^1 = X \cup \{s_{new}\}$; $U^1 = U \cup \{(z, s_{new})\} \cup \{(s_{new}, x) \mid z \in Q^+(z)\}$.
- $\mathbf{G}^2$: $X^2 = X \cup \{s_{new}\}$; $U^2 = U \cup \{(v, s_{new}) \mid v \in Q^-(z)\} \cup \{(s_{new}, x) \mid x \in Q^+(z)\}$.

Cas 4 $s_{nl} \in Q^+(z)$ et $s_{nr} \in Q^+(z)$:
- $\mathbf{G}^2$: $X^2 = X \cup \{s_{new}\}$; $U^2 = U \cup \{(v, s_{new}) \mid v \in Q^-(z)\} \cup \{(s_{new}, x) \mid x \in Q^+(z)\}$.

En plus, dans tous les cas, s_{new} est ajouté à l'orbite $\mathcal{O}$. □

La définition 8.4.6 montre comment la procédure `LookForGraphAlternative` construit les nouveaux graphes lorsqu'une orbite est réduite à un nœud. Dans le premier cas, il faut tester si le nœud s_{nr} est la porte d'entrée de l'orbite représentée par z et le nœud s_{nl} est la porte de sortie. Dans ce cas, trois nouveaux graphes sont proposés: l'un avec s_{new} comme premier nœud de l'orbite, l'autre s_{new} comme le dernier nœud de l'orbite et l'autre avec s_{new} placé en disjonction par rapport aux autres nœuds de l'orbite.

Par exemple, étant donné $E = a^*$, $w = aa$ et $w' = ana$, les deux graphes construits représentent les expressions régulières $(n!a)^*$, $(a\ n!)^*$ et $(a|n)^*$.

Le deuxième cas vérifie si $s_{nl} \in Q^-(z)$ et $s_{nr} = z$. Si c'est le cas, alors deux nouveaux graphes sont aussi proposés : le premier est similaire au premier graphe proposé dans le premier cas ci-dessus et le deuxième correspond au nouveau nœud s_{new} placé en disjonction par rapport par rapport au nœud z.

Par exemple, étant donné $E = ab^*$, $w = abb$ et $w' = anbb$, G^1 corresponde à $a(n!b)^*$ et G^2 à $a(n|b)^*$.

Le troisième cas est symétrique au deuxième. Le quatrième cas corresponde à construction d'un nouveau graphe où le nouveau nœud est en disjonction par rapport au nœud z lorsque $s_{nl} \in Q^-(z)$ et $s_{nr} \in Q^+(z)$.

Par exemple, étant donné $E = a(b\ c)^*d$, $w = ad$ et $w' = and$, le graphe construit G^1 correspond à l'expression régulière $a(n|b\ c)^*d$. On donne en figure 8.10 une représentation graphique de la définition 8.4.6.

Les définitions présentées ci-dessus ne concernent que le cas d'une insertion invalide. Le cas d'une suppression invalide, qui est plus facile à traiter, est présenté ci-dessous.

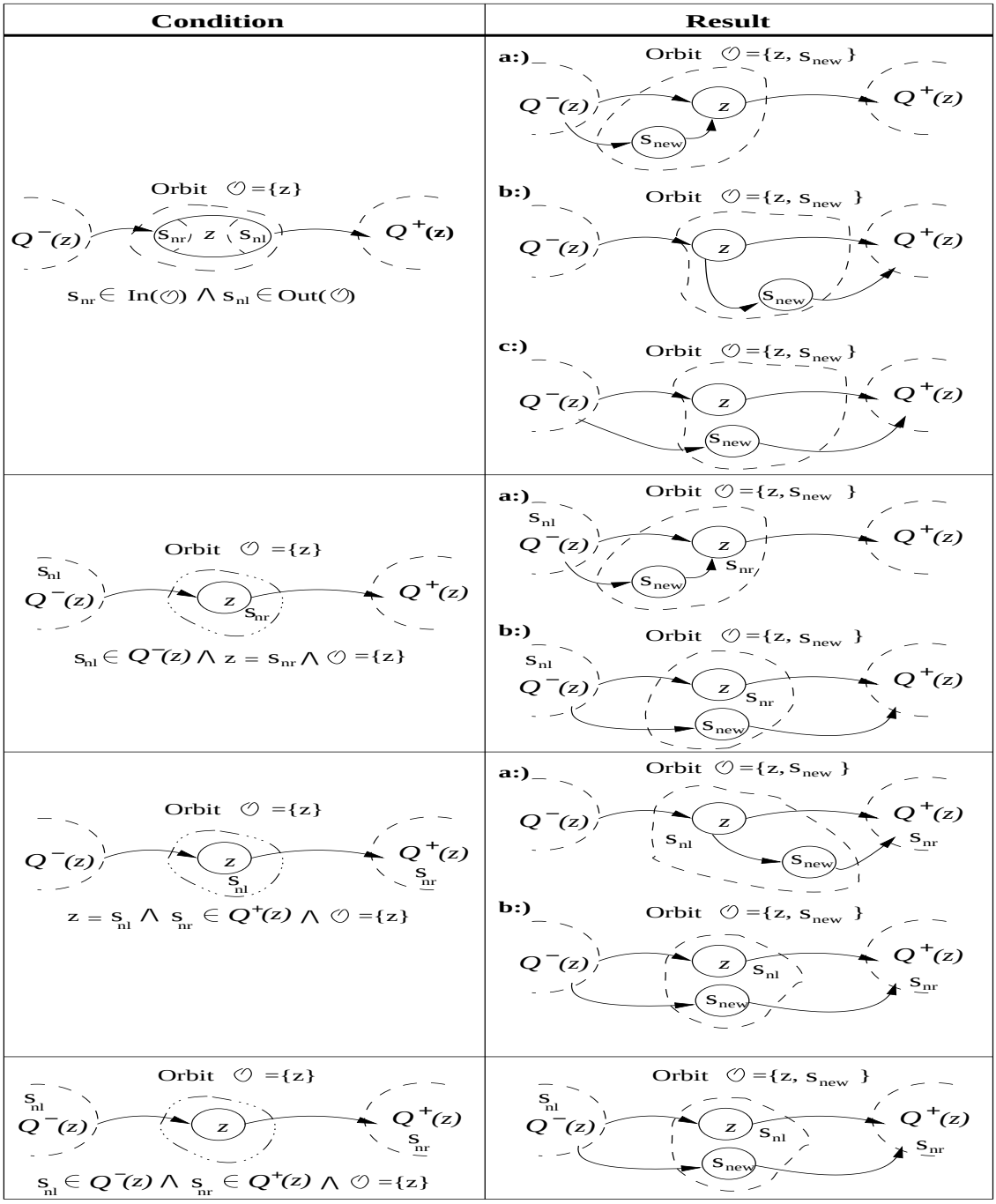


FIG. 8.10 – Conditions à tester et modifications après la réduction d'une orbite.

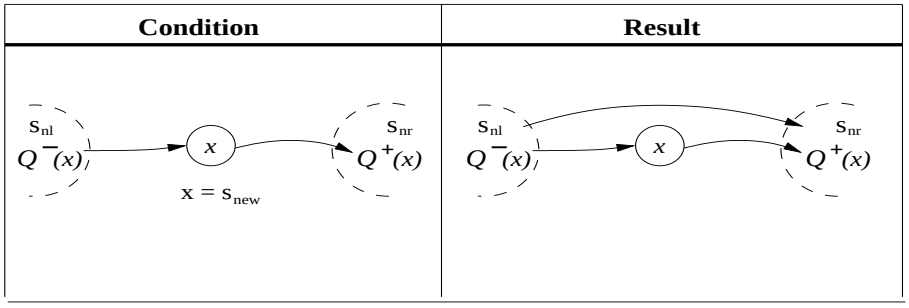


FIG. 8.11 – Modifications à exécuter pour transformer un nœud obligatoire en optionnel

Suppressions invalides

Étant donné une expression régulière E, la tâche de transformer un symbole obligatoire $s \in E$ en un symbole optionnel consiste à transformer l'état qui représente s dans l'automate M_E en un état optionnel. Ainsi, la procédure `LookForGraphAlternative` est appelée de la même façon, en prenant en compte le fait que s_{new} représente déjà un état (ou un nœud) dans le graphe. La définition ci-dessous résume la méthode pour transformer un nœud donné obligatoire en un nœud optionnel.

Définition 8.4.7 Candidat généré à partir d'une suppression invalide : Soit E une expression régulière non ambiguë. Soient $G_1 = (X_1, U_1)$ le graphe de Glushkov qui représente l'automate construit à partir de E et $x \in X_1$ un nœud sur lequel la règle $\mathbf{R}_1$ ou la règle $\mathbf{R}_2$ peut être appliquée. Nous définissons un nouveau graphe $G^1 = (X^1, U^1)$ à partir de G_1 comme suit : si $x = s_{new}$, alors

$$\mathbf{G}^1 : X^1 = X_1;\ U^1 = U_1 \cup \{(y, z) \mid y \in Q^-(x), z \in Q^+(x)\}. \qquad \square$$

Lorsque `GREC` traite une suppression, les seules règles concernées sont les règles $\mathbf{R}_1$ et $\mathbf{R}_2$ car la règle $\mathbf{R}_3$ n'est appliquée qu'aux nœuds déjà optionnels.

Le but de la définition 8.4.7 est de transformer un nœud obligatoire en optionnel. Étant donné un nœud x qui représente un symbole obligatoire dans une expression régulière donnée, x est rendu optionnel en ajoutant un nouveau arc depuis les prédécesseurs de x vers les successeurs de x. Par exemple, étant donné $E = ab^+c$, $w = abc$ et $w' = ac$, le nouveau graphe représente l'expression régulière ab^*c. On donne en figure 8.11 une représentation graphique de la définition 8.4.7.

Notons que les mises à jour apportées aux graphes doivent être accompagnées par des mises à jour de leurs orbites car les orbites des graphes modifiés doivent respecter les propriétés des graphes de Glushkov. La section suivante décrit l'algorithme qui met à jour les orbites lors de l'application des modifications de graphes présentées dans cette section.

8.4.3 Mise à jour des orbites

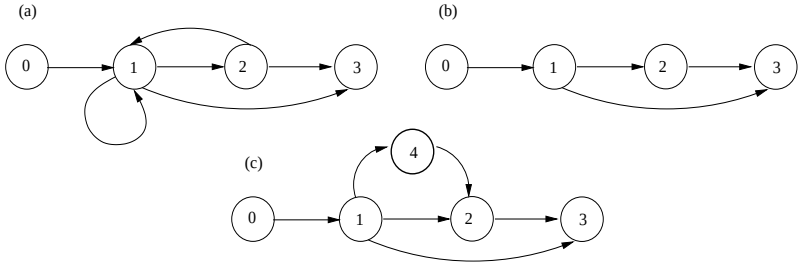
Pendant le processus de construction des expressions régulières candidates, le graphe d'origine G est modifié pour obtenir des nouveaux graphes G^i, qui ont des nouveaux arcs et un nouveau nœud. Il est clair que ces modifications entraînent des modifications dans la hiérarchie des orbites de G^i car les orbites après l'insertion des arcs et du nouveau nœud doivent respecter la relation d'inclusion et surtout les propriétés de stabilité forte et de transversalité forte. Ci-dessous, nous présentons un algorithme utilisé par `LookForGraphAlternative` lorsque les orbites sont mises à jour. Cet algorithme est fondé sur les définitions 8.4.3 à 8.4.7.

Algorithme 8.4.1 - *Mise à jour de la hiérarchie des orbites*

01. Entrée: La paire $(G_1, \mathcal{H})$ et les nœuds s_{nl},s_{nr}, s_{new}, x et y
02. Sortie: La nouvelle hiérarchie des orbites $\mathcal{H}'$
03. $\mathcal{H}' \leftarrow \mathcal{H}$
04. switch (*selon les conditions des définitions 8.4.3 à 8.4.7*) do {
05. Définition 8.4.3 :
06. all cases : Update $\mathcal{O} = \mathcal{O} \cup \{s_{new}\}$ for all $\langle \mathcal{O}, In_{\mathcal{O}}, Out_{\mathcal{O}} \rangle$ in $\mathcal{H}'$
07. such that $s_{nl} \in \mathcal{O}$ and $s_{nr} \in \mathcal{O}$
08. *updateGates*$(\mathcal{H}', G_1)$
09. Définition 8.4.4 :
10. case 1 : Update $\mathcal{O} = \mathcal{O} \cup \{s_{new}\}$ for all $\langle \mathcal{O}, In_{\mathcal{O}}, Out_{\mathcal{O}} \rangle$ in $\mathcal{H}'$
11. such that $s_{nl} \in \mathcal{O}$
12. case 2 : Update $\mathcal{O} = \mathcal{O} \cup \{s_{new}\}$ for all $\langle \mathcal{O}, In_{\mathcal{O}}, Out_{\mathcal{O}} \rangle$ in $\mathcal{H}'$
13. such that $s_{nr} \in \mathcal{O}$
14. case 3 : Update $\mathcal{O} = \mathcal{O} \cup \{s_{new}\}$ for all $\langle \mathcal{O}, In_{\mathcal{O}}, Out_{\mathcal{O}} \rangle$ in $\mathcal{H}'$ such that
15. $x \in \mathcal{O}$ and $y \in \mathcal{O}$ and $s_{nl} \in Q^-(x) \cap Q^-(y)$ and $s_{nr} \in Q^+(x) \cap Q^+(y)$
16. *updateGates*$(\mathcal{H}', G_1)$
17. Définition 8.4.5 :
18. all cases : Update $\mathcal{O} = \mathcal{O} \cup \{s_{new}\}$ for all $\langle \mathcal{O}, In_{\mathcal{O}}, Out_{\mathcal{O}} \rangle$ in $\mathcal{H}'$
19. such that $x \in \mathcal{O}$
20. *updateGates*$(\mathcal{H}', G_1)$
21. Définition 8.4.6 :
22. cases 1,3 : Update $\mathcal{O} = \mathcal{O} \cup \{s_{new}\}$ for all nodes $\langle \mathcal{O}, In_{\mathcal{O}}, Out_{\mathcal{O}} \rangle$ in $\mathcal{H}'$
23. such that $s_{nl} \in \mathcal{O}$
24. case 2 : Update $\mathcal{O} = \mathcal{O} \cup \{s_{new}\}$ for all $\langle \mathcal{O}, In_{\mathcal{O}}, Out_{\mathcal{O}} \rangle$ in $\mathcal{H}'$
25. such that $s_{nr} \in \mathcal{O}$
26. *updateGates*$(\mathcal{H}', G_1)$.
27. Définition 8.4.7 :
28. all cases : *updateGates*$(\mathcal{H}', G_1)$.
29. }

La fonction *updateGates* utilisée, dans l'Algorithme 8.4.1, met à jour les portes d'entrées et sorties de chaque orbite dans le graphe donné. Cette fonction prend en considération les rôles des nœuds s_{nl} et s_{nr} dans les orbites pour ajouter le nouveau nœud comme une porte, si cela doit être le cas. Par exemple, si s_{new} fait partie d'une orbite $\mathcal{O}$ et s_{new} a un arc vers un nœud x et $x \notin \mathcal{O}$, alors s_{new} doit être une porte de sortie de $\mathcal{O}$.

Exemple 8.4.3 Soient $E = (ab?)^+c$ ($\overline{E} = (1\ 2?)^+3$) une expression régulière et $w = abac$ un mot tel que $w \in L(E)$. La figure ci-dessous montre (a) le graphe de Glushkov G construit à partir de l'automate de Glushkov de E et (b) son graphe sans orbite G_{wo}. Soit $\mathcal{H} = \langle(\{1,2\},\{1\},\{1,2\}), (\{0,1,2,3\},\emptyset,\emptyset)\rangle$ la hiérarchie des orbites de G. Étant donné le mot $w' = aba\mathtt{x}c$ ($w' \notin L(E)$) qui représente l'insertion de `x` dans w dans la position 4, le graphe (c) ci-dessous représente l'application de la définition 8.4.5, graphe G^1, sur G_{wo} avant de l'application de la règle $\mathbf{R}_3$ sur le nœud 2.



Dans ce contexte, l'algorithme 8.4.1 est exécuté après l'insertion du nœud 4 (s_{new}) dans G_{wo} comme suit :

- Les lignes 17 à 20 sont exécutées : la définition 8.4.5 a été appliquée.
- Le nouveau nœud est inséré dans toutes les orbites dans lesquelles 2 fait partie. On obtient ainsi $\mathcal{H}' = \langle(\{1,2,4\},\{1\},\{1,2\}),(\{0,1,2,3,4\},\emptyset,\emptyset)\rangle$.
- Ensuite, les portes des orbites mises à jour sont, à leurs tour, mise à jour par la fonction *updateGates*. Comme le nouveau nœud 4 a un arc vers une porte de sortie d'une orbite (nœud 2) et le nœud 2 est optionnel, alors le nœud 4 devient aussi une porte de sortie. Le nœud 4 a aussi un arc entrant depuis une porte d'entrée (nœud 1). Cependant le nœud 1 n'est pas optionnel, alors le nœud 4 ne sera pas une porte d'entrée. La nouvelle hiérarchie des orbites est donc : $\mathcal{H}' = \langle(\{1,2,4\},\{1\},\{1,2,4\}),(\{0,1,2,3,4\},\emptyset,\emptyset)\rangle$.

Si la réduction est appliquée sur la paire $(G^1,\mathcal{H}')$, on obtient l'expression régulière $0(1\ 4?\ 2?)^+3$, donnant $(ax?b?)^+c$. □

8.5 Caractéristiques de GREC

Dans cette section on montre d'abord comment GREC classe les expressions régulières candidates proposées, et comment GREC traite l'ambiguïté des expressions régulières candidates. On présente ensuite une discussion sur la complexité et la raison des structures choisies et, pour terminer, on caractérise les solutions proposées par GREC.

8.5.1 Classification des résultats

Comme montré dans l'exemple 8.2.1, GREC classe les expressions candidates par niveau d'insertion du nouveau symbole. Le classement est fait selon l'orbite utilisée pendant le processus de réduction. Pour que les résultats de GREC soient présentés d'une façon structurée aux utilisateurs, on utilise la notion de *contexte* pour décrire les orbites.

Nous définissons un contexte $\mathcal{C}$ comme un ensemble de symboles qui font partie d'une orbite $\mathcal{O}$ de la hiérarchie des orbites d'un graphe de Glushkov G. Ainsi, pour chaque orbite dans la hiérarchie des orbites, un contexte est défini. Rappelons que chaque nœud interne d'une hiérarchie des orbites $\mathcal{H}$ représente une sous-expression étoilée de l'expression régulière E utilisé pour construire G [CF03, BW92].

Définition 8.5.1 Contextes d'une expression régulière : Soient E une expression régulière et $\overline{E}$ l'expression régulière indicée correspondante. Soient M_E l'automate de Glushkov construit à partir de $\overline{E}$ et $G=(X,U)$ le graphe correspondant. Soit $\mathcal{H}$ la hiérarchie des orbites construite à partir de G. Les *contextes de* E sont construits comme suit :
(i) pour chaque orbite $\mathcal{O} \in \mathcal{H} \setminus \mathcal{O}_r$ (rappelons que $\mathcal{O}_r$ est la racine de $\mathcal{H}$) et pour toutes les positions p dans $\mathcal{O}$, $\mathcal{C}_{\mathcal{O}} = \{\chi(p) \mid p \in \mathcal{O} \wedge \nexists \mathcal{O}_1 \subset \mathcal{O}$ tel que $p \in \mathcal{O}_1\}$.
(ii) Un contexte, appelé *contexte général*, est construit à partir de $\mathcal{O}_r$ comme $\mathcal{C}_{general} = \{\chi(p) \mid p \in \mathcal{O}_r \wedge \nexists \mathcal{O} \subset \mathcal{O}_r$ tel que $p \in \mathcal{O}\}$. □

Comme la définition 8.5.1 le montre, un contexte est formé d'un ensemble de symboles qui appartiennent à une orbite. Les symboles qui n'appartiennent à aucune orbite forment le *contexte général*. Le nombre de contextes d'une expression régulière E est le nombre de ses sous-expressions étoilées plus 1 si E a des symboles qui n'appartiennent à aucune sous-expression étoilée. Il est clair que le nombre de sous-expressions étoilées de E correspond au cardinal de la hiérarchie des orbites (*i.e.* l'ensemble $\mathcal{H}$). La seule différence entre l'ensemble $\mathcal{H}$ et l'ensemble des contextes est qu'un symbole ne peut apparaître que dans un seul contexte (*i.e.*, les contextes sont des ensembles disjoints) alors qu'un symbole peut apparaître dans plusieurs orbites.

Exemple 8.5.1 Considérons l'expression régulière de l'exemple 8.3.2, *i.e.*, $E = (a\ (b|c)^*)^*d$. Dans $\mathcal{H}$ nous trouvons les orbites $O_1 = \{1,2,3\}$ et $\mathcal{O}_3 = \{2,3\}$ qui représentent les sous-expressions étoilées $(a(b|c)^*)^*$ et $(b|c)^*$, respectivement. Les symboles b et c (correspondant aux positions 2 et 3) apparaissent dans les orbites $\mathcal{O}_1$ et $\mathcal{O}_2$, ils seront considérés, selon notre définition, dans un seul contexte. Les contextes sont $\{a\}$ défini à partir de $\mathcal{O}_1$, $\{b,c\}$ défini à partir de O_3 et $\{d\}$ qui est le contexte général. □

La notion de contexte est utile pour présenter les solutions à l'administrateur. Le nombre de solutions construites par `GREC` dépend du nombre d'orbites où le nouveau symbole peut être inséré, et des conditions présentées dans les définitions 8.4.3 à 8.4.6.

Par exemple, étant donné l'expression régulière $E = Shop(\ Invoice(Item\ Price)^*)^*\ Manager$, $s_{nl} = 4$, $s_{nr} = 5$ et $s_{new} = 6$ (correspondant, par exemple, au symbole *Description*), `GREC` retourne, au moins, huit candidats : trois candidats par orbite (s_{nl} participe à deux orbites) et deux candidats construits en dehors des orbites.

Si on prend l'orbite $\{3,4\}$ (la sous-expression étoilée $(Item\ Price)^*$), les candidates sont $Shop(\ Invoice(Item\ Price\ Description?\ )^*)^*Manager$, $Shop(Invoice(Item\ Price\ Description^*\ )^*)^*Manager$ et $Shop\ (\ Invoice\ (\ Item\ Price\ |\ Description\ )^*)^*Manager$. Ces candidats sont stockés dans le contexte correspondant à cette orbite. Ainsi, on présente les candidats divisés en trois contextes ($\{Item, Price\}$, $\{Invoice\}$ et $\{Shop, Manager\}$) et l'administrateur, selon sa connaissance de l'application, pourra choisir le contexte dans lequel se trouve l'option qui lui convient le mieux.

Cette classification des candidats est simple à réaliser : chaque nœud dans la hiérarchie des orbites appartient à un contexte et ainsi, lorsqu'une orbite est utilisée dans le processus de réduction, les candidats construits sont classés dans le contexte qui représente l'orbite utilisée.

Nous remarquons que pour les suppressions, `GREC` construit seulement un candidat et donc il n'y a q'un contexte.

Comme nous l'avons indiqué au chapitre 7, les expressions régulières qui apparaissent dans les règles de transition de notre automate d'arbre sont non ambiguës (*i.e.*, les automates d'états finis associés sont déterministes). Pour préserver la non ambiguïté, `GREC` doit construire des candidats non ambigus. La section suivante présente la stratégie pour préserver la non ambiguïté du schéma résultant de l'évolution.

8.5.2 Ambiguïté

Le W3C préconise que les expressions régulières dans les modèles de contenu des éléments soient non ambigus et il a été montré [Aho97, CZP98] que toute expression régulière peut se réécrire dans une forme non ambiguë. Nous considérons donc que les expressions régulières dans les règles de transitions de notre automate d'arbre sont non ambiguës. Toutefois, la méthode de construction de candidats peut construire un candidat ambigu à partir d'une expression régulière non ambiguë. La proposition suivante présente les contextes dans lesquels une expression régulière candidate ambiguë peut être construite à partir d'une expression régulière non-ambiguë.

Proposition 8.5.1 *Soient E une expression régulière et $G = (X,U)$ le graphe de Glushkov construit à partir de E. Soit* `s` *un symbole inséré ou supprimé du mot $w \in L(E)$ donnant w' tel que $w' \notin L(E)$. Soit $\chi(p)$ l'application qui fait correspondre une position de $Pos(E)$ à son symbole dans E. Soit* `S` *l'ensemble des successeurs de s_{nl} dans G_{wo} défini comme suit:* `S`$= \{\chi(p) \mid p \in Q^+(s_{nl})\}$. `GREC` *construit une expression régulière candidate ambiguë si : (i) le symbole inséré* `s` *appartient à l'ensemble* `S` *ou (ii) le symbole supprimé* `s` *a un successeur ayant la même étiquette.*

Preuve : Une expression régulière E n'est pas ambiguë si son expression indicée $\overline{E}$ ne décrit pas deux mots uxv et uyw tels que $x \neq y$ et $x^{\#} = y^{\#}$(définition 3.3.4).
Nous montrons les conditions dans lesquelles l'expressions régulière non-ambiguë E reste non-ambiguë après une mise à jour :

- Si un symbole a est inséré dans E et a n'appartient pas à l'alphabet de E, la condition de non-ambiguïté sera toujours respectée car la position de a dans $\overline{E}$ correspondra toujours à un symbole unique.
- Si un symbole a est inséré dans E et il existe un symbole a' tel que $a = a'$, la condition de non-ambiguïté est respectée si a' et a n'appartiennent pas a l'ensemble de successeurs d'un symbole b tel que $b \in E$. Cela s'explique car s'il existe deux indices dans $\overline{E}$ qui représentent des symboles ayant la même étiquette et ces indices ne peuvent pas être précédés par un même indice dans les mots décrit par $\overline{E}$, alors la condition de la définition 3.3.4 est respectée.
- Si un symbole a est rendu optionnel dans E, la condition de non-ambiguïté est respectée si l'ensemble de successeurs de a n'a pas un symbole a' tel que $a = a'$ car les prédécesseurs de a n'auront pas des indices qui représentent des symboles ayant la même étiquette.

Ainsi, pour qu'une expression régulière candidates E' soit ambiguë les conditions suivantes doivent être vérifiées :

1. Si un symbole a inséré dans E appartient à l'ensemble de successeurs S d'un symbole b et il existe un symbole a' tel que $a' \in S$ et $a = a'$, alors la nouvelle expression E' est une expression régulière ambiguë.
2. De la même façon, si un symbole a rendu optionnel dans E a un successeur a' tel que $a = a'$, alors la nouvelle expression E' est une expression régulière ambiguë, car les prédécesseurs de a auront a et a' comme successeurs, étant a et a' deux indices différents dans $\overline{E}$.

On démontre donc que le item (i) de la proposition 8.5.1 est vérifié (item 1 ci-dessus) et que le item (ii) de la proposition 8.5.1 est vérifié (item 2 ci-dessus). □

Exemple 8.5.2 Étant donné l'expression régulière $E = (a(b|c)^*)^*d$ (l'exemple 8.3.2) et le mot $w = abcbcd$ dans $L(E)$. Si une insertion du symbole d transforme le mot w en $w' = adbcbcd$, alors $w' \notin L(E)$. Donc, nous avons $s_{nl} = 1$, $s_{nr} = 2$ and $s_{new} = 5$ (remarquons que $\overline{E}$=$(a_1 (b_2|c_3)^*)^*d_4$). Dans ce cas, en utilisant la définition 8.4.4 cas 2 par exemple, `GREC` construit les candidats $E_1 = (a(d?b|c)^*)^*d$ et $E_2 = (a(d^*b|c)^*)^*d$ qui sont ambigus. En effet, s_{nl} qui représente a dans E a déjà un successeur étiqueté d donc, si nous ajoutons un autre successeur étiqueté d, l'expression régulière résultante est ambiguë (*i.e.*, l'automate correspondant à E_1 ou E_2 est non déterministe). □

Lorsque `GREC` identifie que des candidats seront ambigus (en utilisant la propriété proposée dans la proposition 8.5.1), il le signale à l'utilisateur, lequel peut alors appeler une fonction pour transformer l'expression régulière candidate ambiguë en une expression régulière non ambiguë équivalente en appliquant par exemple les approches proposées dans [Aho97, CZP98].

8.5.3 Caractérisation des solutions proposées par `GREC`

Rappelons que, après chaque modification du graphe faite par `GREC` (figure 8.6), l'algorithme `GraphToRegExp` est appliqué. Cela peut être fait car toutes les modifications faites sur G_1 pour obtenir G_{new} respectent les propriétés d'un graphe de Glushkov. C'est ce que formule le théorème ci-dessous.

Théorème 8.5.1 *Soient G un graphe sans orbite réductible et $\mathcal{H}$ la hiérarchie des orbites de G. Soit R_i l'une des règles de réduction $\boldsymbol{R}_1$, $\boldsymbol{R}_2$ ou $\boldsymbol{R}_3$. Pour tous les nœuds s_{nl}, s_{nr} et s_{new}, les graphes G_{new} résultants de l'exécution de* `LookForGraphAlternative`*(G, $\mathcal{H}$, R_i, s_{nl}, s_{nr}, s_{new}) sont réductibles.*

Preuve : Voir annexe C □

Par ailleurs, on peut établir que `GREC` retourne au moins une expression régulière E' différente de l'expression d'origine E et que pour l'insertion E' a au plus un nouveau symbole par rapport à E, *i.e.*, $\mathcal{D}(E, E') \leq 1$. De plus, les langages associés aux solutions données par `GREC` contiennent au moins le langage d'origine et le nouveau mot. En d'autres termes, `GREC` trouve des solutions (au moins une) correctes par rapport aux objectifs énoncés.

Théorème 8.5.2 *Soient E une expression régulière et $L(E)$ le langage associé à E. Étant donné $w[0:n] \in L(E)$ $(0 \leq n)$, soit $w_{ins}[0:n+1]$ (respectivement $w_{del}[0:n-1]$) un mot résultat d'une opération d'insertion (respectivement d'une opération de suppression) sur une position p (avec $0 \leq p \leq n$) tel que $w_{ins} \notin L(E)$ (respectivement $w_{del} \notin L(E)$). Soient M_E l'automate de Glushkov construit à partir de E et G le graphe obtenu à partir de M_E. Soit $(G_{wo}, \mathcal{H})$ la paire qui représente le graphe sans orbite et la hiérarchie des orbites de G, respectivement. Soit s_{nl} un état dans M_E tel que $s_{nl} = \hat{\delta}(q_0, \alpha)$ (où $\alpha = w[0:p-1]$). Soit s_{nr} un état dans M_E tel que $s_{nr} = \hat{\delta}(q_0, \alpha)$ (où $\alpha = w[0:p+1]$). Soit s_{new} un état qui n'existe pas dans Q pour les insertions ou $s_{new} = \hat{\delta}(q_0, \alpha)$ tel que $\alpha = w[0:p]$ pour les suppressions.*
L'exécution de `GREC` *(G_{wo}, $\mathcal{H}$, s_{nl}, s_{nr}, s_{new}) retourne un ensemble fini et non vide d'expressions régulières candidates $\{E_1, \ldots, E_m\}$. De plus, pour tout $E_i \in \{E_1, \ldots, E_m\}$, nous avons $L(E) \cup \{w'\} \subset L(E_i)$ et $\mathcal{D}(E, E_i) = 1$ pour les insertions et $\mathcal{D}(E, E_i) = 0$ pour les suppressions.*

Preuve : Voir annexe D □

8.5.4 Discussion sur la complexité et implantation

Avant de présenter l'étude de la complexité de `GREC`, nous revenons sur le choix des structures utilisées dans notre méthode.

La méthode de guidage d'évolution incrémentale de schéma s'appuie sur des méthodes qui transforment les expressions régulières en automates d'états finis et vice-versa. Cela s'explique car les modèles de contenu des éléments du langage de schéma utilisé sont modélisés par des expressions régulières et ainsi, vérifier si l'instance d'un élément e respecte le modèle de contenu ϑ de e se fait à l'aide d'un automate d'états finis. De plus, si ϑ n'est pas respecté, notre méthode propose des modèles qui peuvent remplacer ϑ. Pour proposer des modèles de contenu candidats, il faut connaître les informations sur l'échec de la vérification et, dans ce contexte, ces informations peuvent être extraites de l'exécution non-réussie de l'automate d'états finis. Ainsi, nous avons décidé, en utilisant les informations sur l'échec de l'automate, de le changer puis de le transformer en expression régulière correspondante. Ceci de façon à ce que l'expression régulière résultante ait une structure similaire à la structure de l'expression régulière d'origine. La vérification de la similarité est faite par la notion de distance définie au début de ce chapitre (définition 8.4.1) et en respectant les caractéristiques de l'expression régulière d'origine (*e.g.*, nombre et ordre des sous-expressions étoilées).

Par exemple, dans le cadre de l'exemple 8.2.1, des candidats qui acceptent l'ancien langage et le nouveau mot sous la forme $q_{Sujet}\ (q_{Annee}\ (q_{Revue})^+)^*\ q_{Conference}?\ (q_{Annee}\ (q_{Revue})^+)^*$ ou $(q_{Sujet}\ (q_{Annee}\ (q_{Revue})^+)^*) \mid (q_{Sujet}\ q_{Annee}\ (q_{Revue}\ q_{Conference})$ ne sont pas proposés car leurs structures (par exemple, nombre de positions, nombre de sous-expressions étoilées, etc.) ont des différences plus importantes de l'expression d'origine $q_{Sujet}\ (q_{Annee}\ (q_{Revue})^+)^*$ que les candidats proposés par `GREC`.

Ainsi, nous avons étudié trois approches de transformations expression régulière ⇒ automate d'états finis ⇒ expression régulière :

- Les algorithmes standards [HMU01] : nous n'avons pas choisi ces algorithmes car si on prend une expression régulière E et on la transforme en automate d'états finis M_E, lorsque l'on applique la modification sur M_E pour obtenir E', E' peut être différente de E par rapport au nombre de positions, aux expressions étoilées, etc.
- L'algorithme de Thompson : dans [GPWZ01, GPWZ04], les auteurs présentent un algorithme qui transforme une expression régulière en automate (appelé *machine de Thompson*) et un algorithme qui fait la transformation inverse. Il est présenté aussi la caractérisation des machines Thompson qui peut être utilisée pour caractériser les expressions régulières utilisées. Cependant, l'un des problèmes d'utiliser cette approche est le résultat de la transformation machine de Thompson en expression régulière : l'ordre des sous-expressions étoilées peut être différent [GPWZ01].
- L'algorithme de Glushkov : cet algorithme transforme une expression régulière en un automate d'états finis (voir sections 7.2 et 8.3.1) et il existe de nombreux travaux qui présentent des méthodes d'implantation de l'algorithme de Glushkov [BS86, BK93, ZPC97]. Dans [CZ00], une caractérisation complète des automates de Glushkov est donnée et un algorithme qui transforme un automate de Glushkov en une expression régulière qui préserve la structure de l'expression d'origine est proposé.

Nous avons choisi l'algorithme de Glushkov et le processus de réduction de [CZ00] car les propriétés des automates de Glushkov nous permettent de caractériser notre méthode (théorèmes 8.5.1 et 8.5.2) et de nous appuyer sur la méthode de construction d'une expression régulière pour proposer les candidats. Ainsi, dans le contexte d'application de la méthode pour guider l'évolution, il faut insérer juste un nouveau symbole[5] dans E et, cela signifie qu'il faut insérer un nouvel état dans l'automate M_E. L'algorithme proposé dans [CZ00] construit une expression régulière à partir d'un automate de Glushkov ayant un nombre de symboles égal au nombre d'états de l'automate (hormis l'état initial). Ainsi, notre méthode propose seulement des expressions régulières E' telles que $\mathcal{D}(E, E') = 1$.

Il existe d'autres approches de transformation d'expression régulière en automate, par exemple l'algorithme de McNaughton et Yamada [CP92], néanmoins ces approches ne présentent pas un algorithme pour faire la transformation inverse et ainsi, il faut utiliser d'autres algorithmes standard de transformation et on tombe dans le problème de structures différentes entre les expressions régulières.

Dans la suite, nous présentons l'étude de la complexité de notre méthode. D'abord nous allons présenter la complexité de la méthode de réduction proposée dans [CZ00].

Construction des orbites maximales : Soit $G = (X, U)$ un graphe (orienté) de Glushkov. Soient $x = |X|$, $u = |U|$ et $\kappa = |X| + |U|$. La complexité de la recherche des composantes fortement connexes est $O(\kappa)$. La construction de la hiérarchie des orbites maximales $\mathcal{H}$ du graphe sans cycle G_{wo} à partir de G peut être résumée comme suit:

1. Nous recherchons les composantes fortement connexes de G pour construire $\mathcal{H}$. Les composantes fortement connexes qui ont plus d'un noeud ou un noeud qui est une boucle sont des orbites maximales de G.
2. Nous calculons les portes de sorties et d'entrées de chaque orbite maximale et ensuite, les arcs (*portes de sorties*, *portes d'entrées*) de chaque orbite sont supprimés.
3. Nous appliquons la recherche des composantes fortement connexes dans les sous-graphes qui représentent les orbites maximales.

[5] Remarquons que l'insertion d'un nouveau symbole dans E veut dire que $\overline{E}$ a une position de plus.

4. Nous revenons au pas 2 ci-dessus jusqu'à ne plus avoir des orbites maximales dans le graphe modifié.
5. Nous avons G_{wo} et $\mathcal{H}$.

Soit i le nombre de fois que les items ci-dessus sont exécutés pour construire G_{wo} et $\mathcal{H}$ à partir de G. Soit x_j (avec $x_j < x$) le nombre de nœuds dans les orbites maximales trouvé à chaque pas $j \leq i$ de la construction de $\mathcal{H}$. Soit u_j (avec $u_j < u$) le nombre des arcs (*portes de sorties*, *portes d'entrées*) de chaque orbite construite dans le pas j. Soit $\kappa_j = x_j + u_j$.
Alors, la complexité en temps d'exécution de construction de G_{wo} et $\mathcal{H}$ est en

$$O(\kappa + (\sum_{j=1}^{j\leq i} \kappa_j)).$$

Réduction de G : Le processus de réduction d'un graphe de Glushkov $G = (X, U)$ prend $G_{wo} = (X', U')$ et $\mathcal{H}$ comme paramètres d'entrés (avec $|X| = |X'|$ et $|U'| \leq |U|$). À chaque pas du processus de réduction, nous vérifions, pour chaque noeud dans G_{wo}, les règles de réduction à appliquer ($\mathbf{R}_1$, $\mathbf{R}_2$ et $\mathbf{R}_3$). Les nœuds qui représentent l'état initial (le noeud 0) et l'état final (le noeud qui représente #) ne sont pas utilisés pendant le processus de recherche de noeuds à réduire (ils sont trouvés en examinant les autres noeuds du graphe) et donc le nombre de noeuds à tester est $x - 2$. Le dernier test est fait lorsque $x = 2$. La décoration d'un noeud avec l'opérateur $^+$ est exécutée de façon directe car nous utilisons les orbites pour guider le processus de réduction : le test de décoration $^+$ est fait lorsqu'il n'existe plus de nœuds à chercher en utilisant une orbite $\mathcal{O}$.
Remarquons que :
(i) les règles $\mathbf{R}_1$ et $\mathbf{R}_2$ fusionnent deux noeuds dans le graphe d'origine, et ainsi, à chaque application de ces règles le graphe a un noeud de moins et
(ii) la règle $\mathbf{R}_3$ supprime un arc du graphe (nous avons donc un arc de moins).
Sans perte de généralité, nous considérons qu'à chaque pas de la réduction, le graphe à réduire a $x - 1$ noeuds. Alors, la complexité en temps d'exécution de la réduction est en :

$$O(((x-2)\times 3) + (((x-2)-1)\times 3) + \ldots + (2\times 3))$$

qui peut être modélisée comme suit (étant $r = x - 2$) :

$$O((\sum_{p=2}^{p\leq r} p)\times 3)$$

développée :

$$O((\frac{r(r-1)}{2})\times 3)$$

et encore développée :

$$O(r(r-1)\times 1,5)$$

On somme la complexité de réduction et la complexité de construction de G_{wo} et $\mathcal{H}$:

$$O(r(r-1)\times 1,5) + (\kappa + (\sum_{j=1}^{j\leq i} \kappa_j)))$$

En sachant que κ est le nombre des arcs et des nœuds, x le nombre de nœuds et r est le nombre de nœuds moins 2 de G, nous pouvons résumer la complexité de la réduction en écrivant $O(x^2)$, c'est-à-dire, dans le cas le pire, la méthode est quadratique en nombre de nœuds du graphe à réduire.

Construction de candidats : La réduction du graphe d'origine est utilisée pour construire des candidats pour le remplacer. Ainsi, la complexité de la construction des candidats est ajoutée à la complexité de la réduction du graphe d'origine.

Pour chaque graphe construit G' à partir du graphe en cours de réduction, il faut compter la complexité de réduction de G' (la hiérarchie des orbites de G' est une copie de la hiérarchie des orbites du graphe d'origine)

Soit m le nombre de graphes à réduire. Soient $G_n = (X_n, U_n)$ le n-ème graphe à réduire ($1 \leq n \leq m$), $x_n = |X_n|$, $u_n = |U_n|$ et $\kappa_n = |X| + |U|$. Soit $r_n = x_n - 2$. Rappelons que $x_n \leq x$, $u_n \leq u$ et $\kappa_n \leq \kappa$ car les graphes G_n à réduire ont moins de nœuds et/ou arcs que le graphe d'origine. La complexité en temps d'exécution de construction des candidats de l'algorithme `GREC` est en

$$O((r(r-1) \times 1,5) + ((r_n(r_n - 1) \times 1.5) \times m) + (\kappa + (\sum_{j=1}^{j \leq i} \kappa_j)))$$

Si nous simplifions la complexité en la considérant quadratique par rapport aux nombre de nœuds du graphe à réduire, nous avons

$$O(x^2 + (x_n^2 \times m))$$

8.6 Exécution pas à pas

Pour terminer ce chapitre, nous présentons un exemple d'exécution pas à pas de `GREC`.

Nous étendons l'exemple 8.2.1 en changeant la règle de transition 8.1 de la façon suivante :

$$Publication, < \emptyset, \emptyset >, q_{Sujet}\ (q_{Annee}(q_{Revue}|q_{Brevet})^+)^* \quad \rightarrow \quad q_{Publication} \tag{8.2}$$

Nous utilisons l'arbre XML t de la figure 7.3. Supposons que l'administrateur veuille insérer un sous-arbre t_p à la position 023 et que l'exécution de $\mathcal{A}$ sur t_p associe l'état $q_{Conference}$ à sa racine. Cette mise à jour déclenche une évolution dans le schéma car le nouveau mot $q_{Sujet}\ q_{Annee}\ q_{Revue}\ q_{Conference}$ n'appartient pas au langage de l'expression régulière de la règle (8.2). Nous rappelons que le mot d'origine était $q_{Sujet}\ q_{Annee}\ q_{Revue}$ (figure 7.3, arbre r).

À partir de l'expression régulière indicée $\overline{E} = q_{Sujet_1}(q_{Annee_2}(q_{Revue_3}|q_{Brevet_4})^+)^*\#_5$ obtenue de la règle (8.2), on enlève les symboles et on ne travaille qu'avec les positions, *i.e.*, on considère $\overline{E} = 1(2(3|4)^+)^*5$. Nous avons le graphe de Glushkov G qui représente $\overline{E}$, son graphe sans orbites G_{wo} (figure 8.12), la hiérarchie des orbites $\mathcal{H} = \{\mathcal{O}_1 = \{3,4\}, \mathcal{O}_2 = \{2,3,4\}, \mathcal{O}_r = \{0,1,2,3,4,5\}\}$ de G et les contextes $\mathcal{C}_1 = \{Revue, Brevet\}$, $\mathcal{C}_2 = \{Annee\}$ et $\mathcal{C}_3 = \{Sujet\}$, définis à partir de $\mathcal{O}_1$, $\mathcal{O}_2$ et $\mathcal{O}_r$, respectivement.

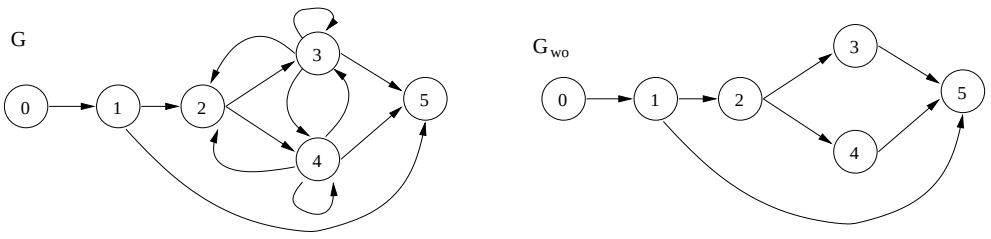


FIG. 8.12 – G : graphe de Glushkov de l'expression régulière $1(2(3|4)^+)^*\ 5$ et G_{wo}.

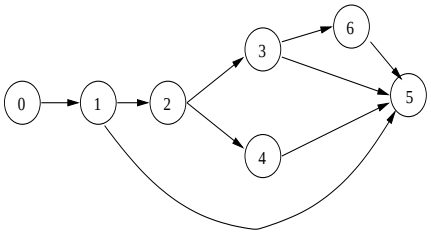
L'exécution de `GREC` (figure 8.6) sur un graphe sans orbite donné (G_1) est divisée en deux parties : la première (la plus externe - lignes 5 et 10) réduit G_1 en utilisant les règles standard de réduction et la deuxième (la plus interne - lignes 7 et 8) modifie G_1 pour obtenir G_{new}, lequel est ensuite transformé en une expression régulière candidate. La construction des nouveaux graphes G_{new} suit les cas montrés dans les définitions 8.4.3 à 8.4.6.

D'abord, considérons les paramètres d'entrée : G_{wo}, $\mathcal{H}$, s_{nl} (le nœud 3 dans G_{wo}), s_{nr} (le nœud 5 dans G_{wo}) et s_{new} (un nœud qui n'appartient pas à G_{wo} que nous notons 6).

Nous considérons les étapes de l'exécution de `GREC` :

1. Comme G_1 (*i.e.*, G_{wo}) a plus d'un nœud, après l'exécution de la commande `if` (ligne 2), `GREC` continue.
2. `ChooseRule` choisit la règle $\mathbf{R}_2$ à appliquer sur les nœuds 3 et 4 dans $\mathcal{O}_1$ (ligne 5).
3. En regardant la première condition présentée dans la définition 8.4.4 on voit que le nœud s_{nl} est affecté par $\mathbf{R}_2$. Donc, la procédure `LookForGraphAlternative` est appelée (ligne 7) et on obtient le graphe G_{new} présenté dans la figure suivante :

   Notons que la hiérarchie des orbites est aussi modifiée pour que le nouveau nœud soit inséré. On a $\mathcal{O}_1 = \{3, 4, 6\}$, $\mathcal{O}_2 = \{2, 3, 4, 6\}$, $In(\mathcal{O}_1) = \{3, 4\}$, $Out(\mathcal{O}_1) = \{3, 4, 6\}$, $In(\mathcal{O}_2) = \{2\}$ et $Out(\mathcal{O}_2) = \{3, 4, 6\}$.

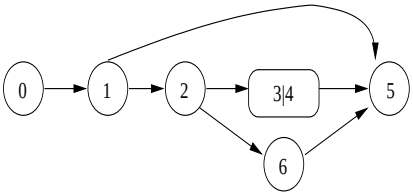
   

   (a) La procédure `GraphToRegExp` reçoit le nouveau graphe et la nouvelle hiérarchie des orbites et retourne l'expression régulière 1(2(3 6?|4)$^+$)*5.

   (b) En utilisant le même raisonnement (pas (3) et (3a) ci-dessus), la procédure `GraphToRegExp` retourne l'expression régulière 1(2(3 6*|4)$^+$)*5 en ajoutant l'orbite {6} (*i.e.* s_{new}) à $\mathcal{H}_{new}$. Les deux expressions régulières sont stockées dans la liste du contexte $\mathcal{C}_1$ car `GREC` travaille avec l'orbite $\mathcal{O}_1$.

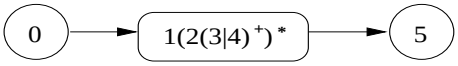
4. `GREC` continue son exécution (ligne 10) et applique la règle $\mathbf{R}_2$ sur le graphe G_1. Ainsi, les nœuds 3 et 4 deviennent le nœud (3|4). Notons que le nouveau nœud représente une orbite complète $\mathcal{O}_1$.
5. La procédure `GREC` est donc rappelée sur ce nouveau graphe.
6. Maintenant, l'orbite à traiter est $\mathcal{O}_2$. La procédure `ChooseRule` choisit la règle $\mathbf{R}_1$ et les nœuds 2 et 3 (ce dernier représentant les anciens nœuds 3 et 4).
7. Comme une orbite vient d'être réduite, la procédure `LookForGraphAlternative` crée des nouveaux graphes selon les modifications présentées dans les définitions 8.4.6 (pour le nœud qui représente maintenant l'orbite $\mathcal{O}_1$) et 8.4.3 (pour la règle $\mathbf{R}_1$). Seule une condition de la définition 8.4.6 est respectée (la troisième condition). Ainsi, on a les graphes G^1_{new} qui donne l'expression 1(2((3|4)6!)$^+$)*5 et G^2_{new} représenté ci-dessous.

   Notons que la hiérarchie des orbites est modifiée pour que les nouvelles expressions régulières soient construites. Les modifications sont : $\mathcal{O}_1 = \{3, 4, 6\}$, $\mathcal{O}_2 = \{2, 3, 4, 6\}$, $In(\mathcal{O}_1) = \{3, 4, 6\}$, $Out(\mathcal{O}_1) = \{3, 4, 6\}$, $In(\mathcal{O}_2) = \{2\}$ et $Out(\mathcal{O}_2) = \{3, 4, 6\}$.

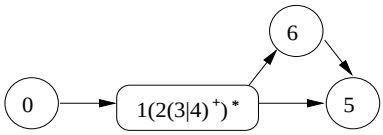
   

   Après avoir obtenu le nouveau graphe, la procédure `GraphToRegExp` est appelée et l'expression calculée est 1(2(3|4|6)$^+$)*5. Les trois expressions calculées dans ce pas sont donc stockées dans la liste du contexte $\mathcal{C}_1$ (correspondant à l'orbite $\mathcal{O}_1$).

8. Avant d'appliquer la règle $\mathbf{R}_1$, l'expression régulière dans le nœud qui représente l'orbite $\mathcal{O}_1$ est décorée avec l'opérateur "+". Après l'application de $\mathbf{R}_1$, le nœud 2 fusionne avec le nœud 3 et on a un nœud avec l'expression $2(3|4)^+$.
9. L'exécution continue et la procédure `GREC` est rappelée (ligne 9).
10. Cette fois la règle à appliquer est la règle $\mathbf{R}_3$. Néanmoins, comme l'orbite $\mathcal{O}_2$ vient d'être réduite, l'algorithme vérifie d'abord si les conditions dans la définition 8.4.6 sont respectées : c'est le cas de la troisième condition, alors le graphe G^1_{new} est proposé et on a l'expression $1(2(3|4)^+6!)^*5$. Le graphe G^2_{new} est aussi proposé et on a l'expression $1((2(3|4)^+)|6)^*5$. Les trois expressions sont stockées dans la liste du contexte $\mathcal{C}_2$ (correspondant à l'orbite $\mathcal{O}_2$). Ensuite, les conditions de la définition 8.4.5 sont testées (le test échoue).
11. Avant d'appliquer la règle $\mathbf{R}_3$ sur G_1, le nœud avec l'expression $2(3|4)^+$ est décoré avec l'opérateur "+" et on obtient $(2(3|4)^+)^+$. Comme $\mathbf{R}_3$ rend un nœud optionnel, l'expression $(2(3|4)^+)^+$ devient $(2(3|4)^+)^*$
12. L'algorithme continue et la règle à appliquer est la règle $\mathbf{R}_1$ sur les nœuds 1 et 2. On obtient le graphe suivant :



13. La règle $\mathbf{R}_1$ est encore applicable. La procédure `LookForGraphAlternative` vérifie si une des conditions de la définition 8.4.3 est respectée.
C'est le cas de la première condition, alors on obtient le graphe ci-contre et l'expression $1(2(3|4)^+)^*6!5$, qui est stockée dans la liste du contexte général $\mathcal{C}_3$ (en dehors de toute orbite).



14. `GREC` continue son exécution jusqu'à ce que G_1 n'ait qu'un seul nœud. Il n'existe alors plus de solutions à proposer.

Les expressions régulières obtenues dans les étapes ci-dessus peuvent maintenant être présentées à l'utilisateur. Avant l'affichage on les traduit dans le langage de schéma d'origine, dans notre cas sous la forme d'une DTD. Par exemple, voici l'affichage des candidats par rapport au contexte $\mathcal{C}_1 = \{Revue, Brevet\}$:

```
1. <!ELEMENT Publication (Sujet,(Annee,((Revue,Conference?)|Brevet)+)*)>
2. <!ELEMENT Publication (Sujet,(Annee,((Revue,Conference*)|Brevet)+)*)>
3. <!ELEMENT Publication (Sujet,(Annee,((Revue|Brevet),Conference?)+)*)>
4. <!ELEMENT Publication (Sujet,(Annee,((Revue|Brevet),Conference*)+)*)>
5. <!ELEMENT Publication (Sujet,(Annee,((Revue|Brevet|Conference)+)*)>
```

qui peuvent remplacer le modèle de contenu d'origine :

```
<!ELEMENT Publications (Sujet,(Annee,(Revue|Brevet)+)*)>
```

Les pas et les résultats de cette exécution ont été faits par un prototype de `GREC`. Ce prototype a été implanté en utilisant le méta environnement ASF+SDF [BHKO02] sous Linux. ASF+SDF est un langage à base des règles qui permet d'implanter des concepts formels. Une fonction en ASF+SDF est exécutée si toutes les fonctions utilisées comme prémisses ont leurs exécution réussites :

```
[implantation cible]
  fonction prémisse 1 (paramètres) = résultat 1
      :
  fonction prémisse n (paramètres) = résultat n
  =============
  fonction cible (paramètres) = résultat
```

Si l'une des prémisses n'est pas exécutée alors la fonction ne sera pas exécutée non plus.

On peut avoir plusieurs implantations de la même fonction, toutefois seulement l'implantation dont les prémisses réussissent leurs exécutions est exécutée[6]. Pour éviter des erreurs d'exécution, une fonction peut être implantée sans prémisses et dans ce cas, elle sera le dernier choix parmi toutes les implantations. Ci-dessous, on montre le morceau principal de l'implantations de `GREC` en ASF+SDF.

La fonction *getCandidates* représente la fonction `GREC` de la figure 8.6 et elle prend comme paramètres d'entrée : $N10 = s_{nl}$, $N20 = s_{nr}$, $AL1 = G_{wo}$, $TN * 1 = \mathcal{H}$, $NT * 1$ est un structure qui stocke le pas de réduction et $LRE * 1$ stocke les candidats calculés. Le retour est un ensemble d'expressions régulières $LRE * 4$.

GraphToRegExp est implanté par la fonction *reduceNodeSet* qui prend les paramètres $N10 = s_{nl}$, $N20 = s_{nr}$, l'orbite à utiliser $TN1$, $AL1 = G_{wo}$, $TN * 1 = \mathcal{H}$, un structure qui stocke le pas de réduction $NT * 1$ et les candidats déjà calculés $LRE * 1$. Elle retourne le nœud qui représente l'orbite réduite $N1$, le nouveau graphe $AL2$, la nouvelle structure qui stocke informations sur la réduction $NT * 2$ et les expressions régulières calculées $LR * 2$.

```
[getCandidates-1]
    reduceNodeSet(N10,N20,TN1, AL1, TN*1,NT*1,LRE*1) = <N1, AL2, NT*2,LRE*2>,
    NT*2 = NT*3 <N1, NRE1, N*4> NT*4,
    TN1 = <N*30,N*40,N*50>,
    deleteNC(N1, N*4) = N*5,
    shrinkOrbits(N1,N*5,TN*1)   = TN*2,
    getCandidates(N10,N20,AL2, TN*2, NT*3 <N1,NRE1,N*4> NT*4,LRE*2) = LRE*4
    ======================================================
    getCandidates(N10,N20,AL1, TN1 TN*1,NT*1,LRE*1) = LRE*4
[getCandidate-2]
    applyAllRulesSS(N10,N20,AL1,NT*1,LRE*1) = <N1, AL2, NT*2,LRE*2>
    ======================================================
    getCandidates(N10,N20,AL1, %empty%, NT*1,LRE*1 ) = LRE*2
[reduceNodes-1]
    reduceNodeSet(N10,N20,TN1, AL1, TN*1,NT*1,LRE*1) = <N1, AL2, NT*2,LRE*2>,
    NT*2 = NT*3 <N1, NRE1, N*4> NT*4,
    TN1 = <N*30,N*40,N*50>,
    deleteNC(N1, N*4) = N*5,
    shrinkOrbits(N1,N*5,TN*1)   = TN*2,
    reduceNodes(N10,N20,AL2, TN*2, NT*2,LRE*2) = NATE1
    ======================================================
    reduceNodes(N10,N20,AL1, TN1 TN*1,NT*1,LRE*1) = NATE1
[reduceNodes-2]
    applyAllRulesSS(N10,N20,AL1,NT*1,LRE*1) = <N1, AL2, NT*2,LRE*2>
    ======================================================
    reduceNodes(N10,N20,AL1, %empty%, NT*1,LRE*1 ) = <N1, AL2, NT*2,LRE*2>
```

Ce chapitre présente notre méthode pour guider l'évolution des schémas XML de façon incrémentale et en préservant la cohérence des documents valides sans les modifier. La méthode proposée dans ce chapitre travaille toujours sur des mots qui ont juste un nouveau symbole ajouté

[6]Si plus d'une implantation peut être candidate à exécution, alors le choix de l'implantation à utiliser ne peut pas être prévu.

ou supprimé. Cela indique que `GREC` peut être utilisé dans un environnement où plusieurs mises à jour sont faites sur un document avant de déclencher le processus de validation. Néanmoins, dans ce cas, l'ensemble de mises à jour à considérer doit respecter une contrainte : les positions des mises à jour ne correspondent jamais à de nœuds frères. Dans ce cas, `GREC` est activé pour chaque mot concerné. Pour les ensembles contenant des mises à jour sur des nœuds frères, une extension de `GREC` est nécessaire. Cette extension sera objet du prochain chapitre.

L'originalité et l'importance de cette méthode sont dues (*i*) à la préservation de la cohérence des documents préexistants par rapport au nouveau schéma sans les modifier, (*ii*) à la construction des candidats qui préserve la structure initial de l'expression régulière d'origine, et (*iii*) à la forme des candidats : si l'expression régulière d'origine est comprise par l'utilisateur, alors les candidates le seront aussi.

Chapitre 9

Extension de GREC aux mises à jour multiples

Dans ce chapitre nous proposons GREC-e (GREC étendu), une extension de GREC dont le but est de proposer des schémas candidats à partir de mises à jour multiples sur un mot w. Nous avons vu au chapitre 8 que les candidats proposées par GREC sont calculés en supposant qu'une seule mise à jour a été faite sur un mot $w \in L(E)$ menant à w' qui peut ne pas appartenir à $L(E)$. En d'autres termes, GREC travaille toujours avec un seul triplet $\langle s_{nl}, s_{nr}, s_{new} \rangle$ pour proposer des candidats pour remplacer le schéma d'origine. Le besoin d'étendre GREC pour guider l'évolution d'un schéma en prenant en considération plusieurs positions dans le nouveau mot est né du constat que l'utilisateur exécute plusieurs mises à jour avant de déclencher le processus de validation. Dans ce contexte, étant donné une suite de n mises à jour, le document résultant de l'application de la i-ème mise à jour ($i < n$) peut être *invalide* par rapport à son schéma. Les conditions de validité peuvent être vérifiées *seulement* à la *fin* d'une suite de mises à jour, c'est-à-dire, après la n-ème mise à jour. Cela introduit une flexibilité dans notre approche qui n'est pas possible si on traite la suite de mises à jour par des appels successifs de GREC. De plus, la méthode des appels successifs ne serait pas efficace car il faudrait alors appeler GREC avec un triplet $\langle s_{nl}, s_{nr}, s_{new} \rangle$, calculer les candidats et, à partir de ces candidats, calculer le triplet suivant et ainsi de suite, jusqu'à ne plus avoir de mises à jour à traiter. Il ne serait pas non plus envisageable de demander plusieurs fois à l'utilisateur de sélectionner une expression régulière intermédiaire avant de pouvoir choisir celle qui représente au mieux le nouveau schéma. GREC-e traite plusieurs positions à la fois, en acceptant un ensemble de triplets $\langle s_{nl}, s_{nr}, s_{new} \rangle$ comme paramètre d'entrée. Ainsi, GREC-e traite des mots qui, après les mises à jour, ont plusieurs positions non reconnues par l'automate M_E. On peut dire que ces mots sont pour la plupart plus distants des mots de $L(E)$ que ne l'étaient les mots traités par GREC.

Le problème traité par GREC-e peut être résumé de la façon suivante :

1. Soit E une expression régulière.
2. Soit w un mot tel que $w \in L(E)$.
3. Soit *UpdateTable* une relation contenant les opérations de mises à jour (insertion, suppression et replacement) effectuées sur w. Chaque n-uplet dans *UpdateTable* indique la position de w où l'opération de mise à jour doit être faite.
4. Soit w' le mot résultant des mises à jour dans *UpdateTable* tel que $w' \notin L(E)$.

GREC-e doit trouver des nouvelles expressions régulières E' telles que $w' \in L(E')$ et $L(E) \subset L(E')$.

9.1 Mises à jour multiples

GREC-e travaille seulement avec les mises à jour effectuées sur un mot composé par la concaténation des états associés à un ensemble de frères dans un arbre XML. En d'autre termes, *UpdateTable* ne contient pas de mises à jour sur n'importe quelle position de l'arbre XML. Chaque appel à GREC-e considère seulement les mises à jour sur un groupe de frères. Ainsi, notre problème reste dans le cadre des automates d'états finis construits à partir des expressions régulières qui modélisent le modèle de contenu d'un élément e.

La relation *UpdateTable* stocke les n opérations de mises à jour sur un mot. Chaque triplet de *UpdateTable* est de la forme (pos, op, s_{pos}), où

- *pos* est la position de mise à jour. Pour simplifier, nous considérons les positions du mot (et non celles de l'arbre).
- *op* est le type de mise à jour à exécuter, *i.e.*, *insert*, *delete* ou *replace*.
- s_{pos} est le symbole (l'état) à insérer dans la position *pos* lorsque *op* est égal à *insert* ou *replace*, sinon $\emptyset$. Dans le cas d'insertion, le symbole est insére toujours à gauche de *pos*. Remarquer que dans notre approche, la position frontière est représentée par le symbole #.

Un mot w' est le résultat des mises à jour dans *UpdateTable* sur un mot w. Le mot w' est construit en appliquant les opérations des mises à jour dans l'ordre qu'elles apparaissent dans *UpdateTable* : $w \xrightarrow{v_1} w_1 \xrightarrow{v_2} w_2 \ldots \xrightarrow{v_n} w'$. Où chaque v_i correspond à une opération de mise à jour *insert*, *delete* ou *replace*. Néanmoins, il faut remarquer que chaque opération v_i décrite par l'utilisateur doit être traduite en une opération interne du système qui prend en compte les mises à jour précédent (c'est-à-dire, qui considère les mises à jour v_j pour $j \leq i$). Cela est nécessaire puisque l'utilisateur décrit ses mises à jour par rapport aux positions du mot w. L'exemple ci-dessous montre l'exécution d'une séquence de mises à jour.

Exemple 9.1.1 Soient t un arbre XML et n un nœud dans t étiqueté *Publications*. Soit *Publication*, $\langle \emptyset, \emptyset \rangle$, $q_{Sujet}\ (q_{Annee}\ {q_{Revue}}^{+})^{*} \rightarrow q_{Publication}$ la règle de transition appliquée sur un nœud étiqueté *Publication*. Soit $w = q_{Sujet}\ q_{Annee}\ q_{Revue}\#$ le mot résultant de la concaténation des états des nœuds fils de n où $w[0] = q_{Sujet}$, $w[1] = q_{Annee}$, $w[2] = q_{Revue}$ et $w[3] = \#$. Supposons que la relation *UpdateTable* contient les opérations de mises à jour sur w ci-dessous. Remarquer que les positions indiquées dans les opérations de mise à jour varient de 0 à $|w| - 1$.

1. $(1, insert, q_{Annee})$
2. $(1, insert, q_{Revue})$
3. $(3, insert, q_{Revue})$

Maintenant, nous analysons l'exécution de cette liste d'opérations de mise à jour, pas à pas :

- La première mise à jour est $(1, insert, q_{Annee})$ sur w donnant $w_1 = q_{Sujet}\ q_{\texttt{Annee}}\ q_{Annee}\ q_{Revue}\ \#$
- La deuxième mise à jour correspond à $(2, insert, q_{Revue})$ sur w_1 donnant $w_2 = q_{Sujet}\ q_{\texttt{Annee}}\ q_{\texttt{Revue}}\ q_{Annee}\ q_{Revue}\ \#$
- La dernière mise à jour correspond à $(5, insert, q_{Revue})$ sur w_2 donnant $w_3 = q_{Sujet}\ q_{\texttt{Annee}}\ q_{\texttt{Revue}}\ q_{Annee}\ q_{Revue}\ q_{\texttt{Revue}}\ \#$

Remarquer que les mots intermédiaires obtenus pendant l'exécution d'une suite de mises à jour peuvent ne pas appartenir à un langage défini par une expression régulière, alors que, à la fin de l'application de toutes les opérations, le mot résultant peut appartenir à langage définit par cette expression régulière. Ainsi, le mot w_1 n'appartient pas au langage défini par l'expression régulière $E = q_{Sujet}\ (q_{Annee}\ {q_{Revue}}^{+})^{*}\ \#$. Néanmoins, le mot w_3 (le mot résultant après l'application des toutes les mises à jour) appartient à $L(E)$. Cela indique que les opérations de mises à jour de la relation *UpdateTable* peuvent être appliquées sans compromettre la validité du document

concerné. Ce n'est donc pas le cadre d'utilisation de GREC-e. En revanche, soit *UpdateTable* la relation qui contient les opérations suivantes :

1. $(1, insert, q_{Annee})$
2. $(1, insert, q_{Revue})$
3. $(3, insert, q_{Conference})$
4. $(3, insert, q_{Brevet})$

La première mise à jour ne préserve pas la validité du mot par rapport au langage décrit par E (*i.e.*, $w_1 = q_{Sujet}\ q_{\texttt{Annee}}\ q_{Annee}\ q_{Revue}\ \#$ n'appartient pas à $L(E)$). La deuxième mise à jour rétablit la validité ($w_2 = q_{Sujet}\ q_{\texttt{Annee}}\ q_{\texttt{Revue}}\ q_{Annee}\ q_{Revue}\ \#$ appartiennent à $L(E)$). Les deux dernières opérations produisent des mots invalides par rapport au langage décrit par E (*i.e.*, $w_3 = q_{Sujet}\ q_{\texttt{Annee}}\ q_{\texttt{Revue}}\ q_{Annee}\ q_{Revue}\ q_{\texttt{Conference}}\ \#$ et $w_4 = q_{Sujet}\ q_{\texttt{Annee}}\ q_{\texttt{Revue}}\ q_{Annee}\ q_{Revue}\ q_{\texttt{Conference}}\ q_{\texttt{Brevet}}\ \#$ n'appartiennent pas à $L(E)$) et donc, l'ensemble de mises à jour est invalide. L'utilisation de GREC-e se fait dans ce genre de situation. □

L'exemple 9.1.1 illustre certains aspects importants de l'extension de GREC. En effet, dans le cadre de mises à jour multiples, il existe des positions mises à jour dans le mot qui ne provoquent pas l'échec de l'automate pendant l'acceptation du nouveau mot. Par conséquent, la méthode pour trouver les états s_{nl}, s_{nr} dans l'automate qui accepte $L(E)$ ne peut pas être la même que la méthode utilisée dans GREC.

De plus, le traitement simultané des suppressions et des insertions apporte une difficulté dans la recherche des états s_{nl} et s_{nr}. Par exemple, supposons qu'un symbole obligatoire d'un mot est supprimé et dans sa position n nouveaux symboles sont insérés. La position d'origine du symbole supprimé n'est plus la même et le mot change de taille. Dans ce cadre, décider si le mot résultat n'est pas accepté par l'automate du fait que des nouveaux symboles ont été insérés ou qu'un symbole a été supprimé n'est pas trivial.

Notre méthode pour plusieurs mises à jour s'appuie sur les informations contenues dans la relation *UpdateTable* pour trouver les positions à traiter dans le nouveau mot. Dans la prochaine section, nous allons présenter les structures auxiliaires utilisées pour construire les triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ utilisés par GREC-e.

9.2 Préparation des données

Dans un premier temps nous nous concentrons sur la préparation des données qui doivent être passées à GREC-e pour la construction des nouveaux schémas.

Soit une expression régulière E, son automate de Glushkov $M_E = (Q, \Gamma, \Delta, 0, F)$ et un mot w tel que $w \in L(E)$. Nous supposons un mot $w' \notin L(E)$ résultant de la mise à jour de w en appliquant les modifications listées dans *UpdateTable*. Pour chaque symbole $w'[i]$ dans $w'[0:n]$ à partir duquel w' n'est plus accepté par l'automate M_E, nous construisons un triplet $TStates = \langle s_{nl}, s_{nr}, s_{new}\rangle$. Remarquer que la méthode qui cherche les triplets $TStates$ est basée sur l'automate d'origine, c'est-à-dire, l'automate qui accepte w. Ainsi, pour trouver les morceaux de w' qui peuvent provoquer l'échec de l'automate nous faisons d'abord la correspondance entre les parties communes de w et de w'. Cette correspondance est stockée dans un tableau appelé $TMaps$. La recherche des parties communes entre les deux mots est guidée par la relation *UpdateTables*. Ainsi, la recherche des triplets $TStates$ est guidée par $TMaps$.

Dans la suite nous allons d'abord présenter l'approche de construction de $TMaps$ et ensuite présenter l'utilisation de $TMaps$ dans la construction des triplets $TStates$.

9.2.1 Construction de $TMaps$

Dans GREC, si le nouveau mot w' (construit à partir d'une seule mise à jour sur un mot w) n'est plus accepté par M_E, alors la position où la mise à jour a été exécutée est l'endroit où M_E a échoué. Dans le cas de mises à jour multiples, plusieurs positions de w' peuvent être la cause d'un échec de M_E et ces positions peuvent ne pas être les positions utilisées dans la relation $UpdateTable$ qui sont des positions dans w.

Par exemple, soient $E = ab^*c$ une expression régulière, $w = abbbc$ un mot valide, $w' = abxc$ un mot non valide résultant des mises à jour $(1, delete, b)$, $(3, delete, b)$ et $(4, insert, x)$ sur w. La position de w' qui fait échouer l'automate M_E est la position 2 (la position de x) et les positions des mises à jour de w sont 1, 3 et 4.

Pour GREC-e, il faut chercher dans les mots concernés, c'est-à-dire, le mot d'origine w et le mot après les mises à jour w', tous les symboles qui sont susceptibles d'engendrer un triplet $TStates$. Le tableau de correspondance $TMaps$ stocke la correspondance entre les symboles du mot d'origine et ceux du nouveau mot. $TMaps$ a deux colonnes, étiquetées old (qui représente le mot d'origine) et new (qui représente le mot mis à jour). Ses lignes stockent la correspondance entre[1] :

1. Les symboles qui existent dans w et dans w' (*i.e.*, $w[i:j] \rightarrow w'[k:l]$) tels que $w[i] = w'[k]$, $w[i+1] = w'[k+1]$, ..., $w[j] = w'[l]$.
2. Les symboles qui existent dans w et n'existent plus dans w' (*i.e.*, $w[i:j] \rightarrow \varepsilon$).
3. Les symboles qui n'existent pas dans w mais qui existent dans w' (*i.e.*, $\varepsilon \rightarrow w'[k:l]$).

Remarquer que le cas 2 représente le cas où les symboles de w ont été supprimés et le cas 3 représente le cas où de nouveaux symboles ont été insérés dans w. Les lignes de $TMaps$ sont organisées par ordre de lecture du mot w, c'est-à-dire, de gauche à droite.

Le tableau $TMaps$ est construit de façon à respecter les propriétés suivantes : (i) toutes les lignes font correspondre des positions qui existent dans w et w', (ii) chaque position dans w et w' a, au maximum, une correspondance et (iii) les correspondances ne se croisent pas, par exemple, si une position i de w fait correspondance à une position j de w', il ne peut pas exister une deuxième correspondance qui fait correspondre une position $k > i$ dans w avec une position $l < j$ de w'.

Exemple 9.2.1 Le tableau $TMaps$ construit pour le mot d'origine $w = q_{Sujet}\ q_{Annee}\ q_{Revue}\ \#$ en utilisant la première relation $UpdateTable$ de l'exemple 9.1.1 est:

	old	new
1	q_{Sujet}	q_{Sujet}
2	ε	$q_{Annee}\ q_{Revue}$
3	$q_{Annee}\ q_{Revue}$	$q_{Annee}\ q_{Revue}$
4	ε	$q_{Annee}\ q_{Revue}$
5	q_{Revue}	q_{Revue}
6	#	#

□

L'algorithme de construction de $TMaps$ utilise le mot d'origine w et la relation $UpdateTable$ (triplets sous la forme (pos, op, s_{pos})) de mises à jour à appliquer sur w. Nous présentons d'abord l'algorithme 9.2.1 qui parcourt w et, en utilisant $UpdateTable$, cherche les positions de w mises

[1] Cette correspondance correspond à celle de la définition des opérations d'édition sur des chaînes de caractères dans [WF74].

à jour. Si une position i de w n'a pas été mise à jour alors le symbole correspondant aura une correspondance avec un symbole dans une position j du mot mis à jour. Néanmoins, si une position i de w a été supprimée (subit une opération de suppression ou remplacement), alors le symbole correspondant doit avoir une ligne du type $w[i] \rightarrow \varepsilon$ dans $TMaps$. Cette situation est contrôlée par la variable logique *upd* dans l'algorithme 9.2.1.

L'algorithme 9.2.1 contrôle la manière dont les symboles sont rangés dans le tableau $TMaps$ et c'est l'algorithme 9.2.2 qui est responsable pour insérer les symboles dans $TMaps$. Ces deux algorithmes sont présentés ci-dessous.

Algorithme 9.2.1 - Construction de $TMaps$

```
Input: w et la liste de mises à jour UpdateTable
Output: TMaps
// Remarquer que le dernier symbole de w est # qui représente la fin de w
for i from 0 to |w| − 1 do
{
  upd = false
  while (∃u ∈ UpdateTable | u = (i, op, s_pos))
  {
    switch (op)
    {
      case delete : buildTable(w[i], ε, TMaps)
                    upd = true
      case insert : buildTable(ε, s_pos, TMaps)
      case replace: buildTable(w[i], ε, TMaps) // suppression suivie par
                    buildTable(ε, s_pos, TMaps) // l'insertion
                    upd = true
    } //end switch
    UpdateTable = UpdateTable \ {u}
  } // end while
  //Vérifie si il y a eu des suppressions des symboles dans le mot d'origine.
  if (¬upd)
  {
    buildTable(w[i], w[i], TMaps)
  }
} // end for i
```

L'algorithme *buildTable* qui insère ou met à jour les lignes dans le tableau $TMaps$ est présenté ci-dessous.

Algorithme 9.2.2 - L'algorithme buildTable

01. `Input :` Un symbole $symb_{old}$ qui existe/existait d'un le mot d'origine.
02. Un symbole $symb_{new}$ inséré dans le mot d'origine.
03. Le tableau $TMaps$.
04. `Output:` Le tableau $TMaps$ mise à jour.
05. `let` n `be the number of rows of` $TMaps$
06. `if` $(symb_{old} = \varepsilon)$ `{ //` la ligne dans $TMaps$ représente une insertion
07. `if` $(TMaps[n, old] = \varepsilon)$ `{`
08. $TMaps[n, new] = TMaps[n, new] \cdot symb_{new}$
09. `} //`
10. `else {`
11. $n = AddNewRow(TMaps)$
12. $TMaps[n, old] = \varepsilon$
13. $TMaps[n, new] = symb_{new}$
14. `}`
15. `} //` *fin du if* $symb_{old} = \varepsilon$
16. `else {`
17. `if` $(symb_{new} = \varepsilon)$ `{ //` la ligne dans $TMaps$ représente une suppression
18. `if` $(TMaps[n, new] = \varepsilon)$ `{`
19. $TMaps[n, old] = TMaps[n, old] \cdot symb_{old}$
20. `}`
21. `else {`
22. $n = AddNewRow(TMaps)$
23. $TMaps[n, old] = symb_{old}$
24. $TMaps[n, new] = \varepsilon$
25. `}`
26. `}`
27. `else { //` la ligne dans $TMaps$ représente une correspondance entre les symboles
28. `if` $(TMaps[n, old] = \varepsilon$ `or` $TMaps[n, new] = \varepsilon$ `) { //` de w et de w'
29. $n = AddNewRow(TMaps)$
30. $TMaps[n, old] = symb_{old}$
31. $TMaps[n, new] = symb_{new}$
32. `}`
33. `else {`
34. $TMaps[n, old] = TMaps[n, old] \cdot symb_{old}$
35. $TMaps[n, new] = TMaps[n, new] \cdot symb_{new}$
36. `}`
37. `}`
38. `}`

L'exemple suivant illustre l'exécution des algorithmes 9.2.1 et 9.2.2.

Exemple 9.2.2 Nous considérons la relation *UpdateTable* de l'exemple 9.1.1, c'est-à-dire, (1, *insert*, q_{Annee}), (1, *insert*, q_{Revue}) et (3, *insert*, q_{Revue}). Ainsi, à partir de $w = q_{Sujet}\ q_{Annee}\ q_{Revue}\ \#$, nous obtenons $w' = q_{Sujet}\ q_{Annee}\ q_{Revue}\ q_{Annee}\ q_{Revue}\ q_{Revue}\ \#$.

1. Initialement, dans l'algorithme, pour $i = 0$, il n'existe aucun triplet dans *UpdateTable*. Alors la boucle *while* n'est pas exécutée et la variable *upd* a toujours la valeur *faux*. La condition dans la ligne 21 est respectée, donc la procédure *buildTable* est appelée avec les paramètres: q_{Sujet}, q_{Sujet} et $TMaps$. Dans la procédure *buildTable*, les conditions des lignes 06 et 17 ne sont pas vérifiées, alors le *else* de la ligne 27 est vérifié. La condition de la ligne

28 est respectée, *i.e.*, $TMaps$ n'a pas de valeur dans la position n (à ce moment $n = 0$), alors une nouvelle ligne est insérée dans le tableau $TMaps$ (la fonction $AddNewRow$ ajoute une ligne à $TMaps$ et retourne la position de cette ligne) :

	old	*new*
1	q_{Sujet}	q_{Sujet}

2. Pour $i = 1$, w a été mise à jour, alors la boucle *while* est exécutée car il existe des triplets dans *UpdateTables* dont $pos = 1$:
   (a) Pour (1, *insert*, q_{Annee}), le *case* de la ligne 13 (algorithme 9.2.1) est exécuté et la procédure *buildTable* est appelée avec les paramètres ε, q_{Revue} et $TMaps$. La condition de la ligne 06 est vérifiée, cependant la condition de la ligne 07 n'est pas vérifiée (la colonne *old*, ligne 1 de $TMaps$ a la valeur q_{Sujet}), alors les lignes 11 à 13 sont exécutées, donnant la deuxième ligne de $TMaps$:

	old	*new*
1	q_{Sujet}	q_{Sujet}
2	ε	q_{Annee}

   Le triplet (1, *insert*, q_{Annee}) est supprimé de *UpdateTable* (ligne 15).
   (b) Dans la deuxième exécution de la boucle *while* pour (1, *insert*, q_{Revue}), le *case* de la ligne 13 est exécuté et la procédure *buildTable* est appelée avec les paramètres ε, q_{Annee} et $TMaps$. La condition de la ligne 06 est vérifiée et aussi la condition de la ligne 07 (la colonne *old*, ligne 2 de $TMaps$ a la valeur ε). Alors la ligne 08 est exécutée, ajoutant q_{Annee} à la ligne 2 de la colonne *new* de $TMaps$:

	old	*new*
1	q_{Sujet}	q_{Sujet}
2	ε	$q_{Annee}\ q_{Revue}$

   Le triplet (1, *insert*, q_{Revue}) est supprimé de *UpdateTable* (ligne 15).
   (c) Il n'existe plus de triplet dans *UpdateTable* dont $pos = 1$, alors la boucle *while* termine.

   Comme il n'y a pas eu des suppressions, la valeur de *upd* reste à $false$. La condition ligne 21 est vérifié et ainsi la procédure *buildTable* est appelée avec les paramètres q_{Annee}, q_{Annee} et $TMaps$. Le *else* de la ligne 27 est exécuté et la condition de la ligne 28 est vérifiée (la colonne *old*, ligne 1 de $TMaps$ est égal à ε). Alors les lignes 29 à 31 sont exécutées, ajoutant la troisième ligne de $TMaps$:

	old	*new*
1	q_{Sujet}	q_{Sujet}
2	ε	$q_{Annee}\ q_{Revue}$
3	q_{Annee}	q_{Annee}

3. Dans l'algorithme 9.2.1, nous avons la variable $i = 2$ (troisième symbole de w). Il n'existe pas un triplet dans *UpdateTable* dont $pos = 2$, alors la boucle *while* n'est pas exécutée. La procédure *buildTable* est appelée avec les paramètres q_{Revue}, q_{Revue} et $TMaps$ (ligne 23). Le *else* de la ligne 27 est exécuté et la condition de la ligne 28 n'est pas vérifiée (aucune colonne de $TMaps$, ligne 2 n'est égal à ε), alors les lignes 34 et 35 sont exécutées, concaténant le symbole q_{Revue} à la troisième ligne de $TMaps$:

	old	*new*
1	q_{Sujet}	q_{Sujet}
2	ε	$q_{Annee}\ q_{Revue}$
3	$q_{Annee}\ q_{Revue}$	$q_{Annee}\ q_{Revue}$

4. Pour $i = 3$, la boucle *while* est exécutée car il existe un triplet dans *UpdateTable* dont $pos = 3$, *i.e.*, (3, *insert*, q_{Revue}). Le *case* de la ligne 13 est exécuté et la procédure *buildTable* est appelée avec les paramètres ε, q_{Revue} et $TMaps$. La condition de la ligne 06 est vérifiée, cependant la condition de la ligne 07 n'est pas vérifiée (la colonne *old*, ligne 3 de $TMaps$ n'est pas vide), alors les lignes 11 à 13 sont exécutées, donnant la quatrième ligne de $TMaps$:

	old	*new*
1	q_{Sujet}	q_{Sujet}
2	ε	$q_{Annee}\ q_{Revue}$
3	$q_{Annee}\ q_{Revue}$	$q_{Annee}\ q_{Revue}$
4	ε	q_{Revue}

Le triplet (3, *insert*, q_{Revue}) est supprimé de *UpdateTable* (ligne 15). La condition de la ligne 21 est vérifiée ($upd = false$) et ainsi la procédure *buildTable* est appelée avec les paramètres #, # et $TMaps$. Le *else* de la ligne 27 est exécuté ainsi nous avons la cinquième ligne de $TMaps$:

	old	*new*
1	q_{Sujet}	q_{Sujet}
2	ε	$q_{Annee}\ q_{Revue}$
3	$q_{Annee}\ q_{Revue}$	$q_{Annee}\ q_{Revue}$
4	ε	q_{Revue}
5	#	#

5. Il n'existe plus de positions à traiter, c'est la fin d'exécution l'algorithme.

Le tableau $TMaps$ construit est le même de l'exemple 9.2.1. □

À la fin du processus de construction du tableau de correspondance $TMaps$, on remarque que :

- Si nous parcourons le tableau $TMaps$ ligne à ligne en concaténant les valeurs de chaque colonne, nous aurons le mot d'origine dans la colonne *old* et le nouveau mot dans la colonne *new*. Par abus de notation, nous pourrons référencer des entrées dans le tableau $TMaps$ parfois comme des positions aussi bien que comme des symboles des mots w et w'.
- Deux lignes successives l et $l+1$ de $TMaps$ ne décrivent pas les mêmes opérations de mises à jour.
- La dernière ligne de $TMaps$ aura toujours des valeurs différentes de ε car nous ajoutons à l'expression régulière E une marque de fin # et, par conséquent, nous ajoutons aussi la marque de fin # aux mots w et w'.

Le résultat de la correspondance entre les mots w et w' dans le tableau $TMaps$ n'est pas minimal, *i.e.*, $TMaps$ pourrait avoir moins de lignes. Cependant $TMaps$ est construit simplement, en temps linéaire par rapport à la taille de w et le nombre d'opérations de mises à jour exécutées sur w. Le problème de correspondance entre chaînes de caractères (ou mots) est un problème étudié dans le domaine de la correction automatique des chaînes de caractères [Tic84, WF74]. Dans ce cadre, on définit la distance entre deux mots A et B à partir de la séquence d'opérations d'édition nécessaires pour transformer A en B. Les opérations d'édition comprennent au moins l'insertion d'un caractère et la suppression d'un caractère. Un coût est associé à chaque opération d'édition. La distance est souvent définie comme la séquence d'opérations d'édition la moins coûteuse. Une façon de calculer cette distance est de calculer les sous-séquences communes de A et B. La résolution de ce problème se fait en $O(\mid A \mid \times \mid B \mid)$ [UAH76]. On pourrait utiliser cette approche pour construire $TMaps$ à partir de w et w', mais pour la suite nous préférons avoir $TMaps$ construit à partir des mises à jour car les informations de mises à jour sont importantes pour calculer les schémas candidats.

Exemple 9.2.3 Étant donné les mots $w = q_{Sujet}\ q_{Annee}\ q_{Revue}\ \#$ et $w' = q_{Sujet}\ q_{Annee}\ q_{Revue}\ q_{Annee}\ q_{Revue}\ q_{Revue}\ \#$, les sous-séquences communes entre w et w' peuvent être :

(i) $q_{Sujet}\ q_{Annee}\ q_{Revue}\ \#$
$q_{Sujet}\ q_{Annee}\ q_{Revue}\ q_{Annee}\ q_{Revue}\ q_{Revue}\ \#$ ou

(ii) $q_{Sujet}\ q_{Annee}\ q_{Revue}\ \#$
$q_{Sujet}\ q_{Annee}\ q_{Revue}\ q_{Annee}\ q_{Revue}\ q_{Revue}\ \#$

Ainsi, dans notre contexte, $TMaps$ aurait seulement 3 lignes, néanmoins les informations des mises à jour ne seraient pas codées dans le tableau de la manière qu'elles ont été exécutées. □

Lorsque le tableau $TMaps$ est disponible, le processus de construction des triplets $TStates = \langle s_{nl}, s_{nr}, s_{new}\rangle$ de GREC-e peut être décrit. La relation qui stocke les triplets $TStates$ est appelée $RSTates$. Dans le cadre des mises à jour multiples, nous pouvons, en utilisant les triplets dans $RSTates$, restreindre la construction des candidates. Pour cela, nous proposons aussi un ensemble appelé $STrans$ qui contient les rapports entre les différents $TStates$ construits. La section suivante présente ces deux structures.

9.2.2 Construction de $RTStates$ et $STrans$

L'algorithme GREC-e propose des candidates qui peuvent remplacer un schéma d'origine en prenant en considération plusieurs triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$. GREC-e utilise les mêmes raisonnements que GREC. Le graphe d'origine est réduit et pendant le processus de réduction, GREC-e vérifie si un nouveau nœud (le symbole représenté par s_{new}) peut être inséré. GREC-e diffère de GREC par les points suivants :

- Avant d'appeler la fonction de réduction d'un graphe en expression régulière, tous les triplets dans $RSTates$ doivent être traités.
- La suppression des symboles est modélisée par un triplet de la forme $\langle s_{nl}, s_{nr}, null\rangle$. Cela veut dire que, tous les nœuds entre s_{nl} et s_{nr} (non-inclus) seront rendus optionnels.
- Des candidats engendrés peuvent être rejetés car GREC-e a un mécanisme qui, basé sur les triplets dans $RTStates$, vérifie si un candidat respecte le mot mis à jour.

Dans l'exemple suivant, nous présentons l'intuition de ce nouveau mécanisme.

Exemple 9.2.4 Soient $w = abc$ un mot, $w' = ab\mathtt{xy}c$ le mot w mis à jour et $E = ab^*c$ ($\overline{E} = 1\ 2^*\ 3$) une expression régulière telle que $w \in L(E)$. Nous considérons $RTStates = (\langle 2, 3, 4\rangle, \langle 2, 3, 5\rangle)$ où l'état 2 représente b, l'état 3 représente c, l'état 4 représente x et le 5 représente y. Supposons maintenant l'utilisation de la définition 8.4.6 dans la construction d'expressions régulières candidates. Comme $RTStates$ a deux triplets, les candidats proposés auront deux nouveaux nœuds (les nœuds qui représentent x et y). Parmi ces candidats, les expressions régulières suivantes sont générées : $E_1 = a(bx!y!)^*c$, $E_2 = a(by!x!)^*c$, $E_3 = ab^*x!y!c$ et $E_4 = ab^*y!x!c$. Les candidats ainsi obtenues ne répondent pas à toutes les conditions de proposition de candidats, *i.e.*, pour chaque expression régulière candidate E', $w' \in L(E')$ et $L(E) \subset L(E')$. En particulier, E_2 et E_4 ne décrivent pas w' car les mots dans leurs langages ne peuvent pas avoir le sous-mot xy. □

L'exemple 9.2.4 montre que dans le cadre de mises à jour multiples, les positions des nouveaux symboles dans le mot mis à jour doivent être considérées lors du processus de construction des candidats. Ainsi, il est important de préciser que les automates d'états finis construits à partir des expressions régulières candidates de l'exemple 9.2.4 doivent avoir une transition[2] $\delta(4, y) \neq \perp$ (où 4 est l'état qui représente le symbole x). Dans ce contexte, les candidats E_2 et E_4 ne seraient

[2]Notons que le symbole $\perp$ est utilisé comme "pseudo-état" pour dénoter l'absence d'une telle transition dans un automate et on considère $\delta(q, \varepsilon) = q$ et $\delta(q, \emptyset) = \perp$.

pas proposés car leurs automates n'ont pas une transition $\delta(4, y) \neq \perp$. Par abus de notation, nous allons appeler une transition $\delta(etat, symb) = \perp$ comme une transition qui amène à un *état invalide* de l'automate (ou une *transition invalide*) et une $\delta(etat, symb) \neq \perp$ comme une transition qui amène à un *état valide* de l'automate (ou une *transition valide*).

L'ensemble $STrans$ est défini pour résoudre cela : les membres de $STrans$ sont des paires $(state, symb)$ pour indiquer que la transition $\delta(state, symb)$ dans les automates des expressions régulières candidates ne doit jamais retourner un état *invalide*. Dans notre approche, on n'attend pas la construction de l'automate de l'expression régulière candidate pour vérifier cette contrainte, elle est vérifiée lors de la construction du graphe candidat.

La construction de $STrans$ est faite en parallèle avec la construction de $RTStates$. L'algorithme ci-dessous présente la boucle principale du processus de construction de $RTStates$ et $STrans$. La construction est guidée par le tableau $TMaps$. Comme nous avons déjà mentionné, si une ligne de $TMaps$ a la valeur ε dans la colonne *old* alors les symboles dans la colonne *new* ont été insérés dans le mot d'origine. Sinon, si la colonne *new* a le valeur ε, alors les symboles dans la colonne *old* ont été supprimés. Si aucun de ces deux cas ne sont vérifiés, alors les symboles dans la colonne *old* ou *new* n'ont pas été changés. Selon les valeurs des colonnes *old* et *new*, l'algorithme traite la construction de $RTStates$ et $STrans$ de manière différente. Remarquer que pour les suppressions l'ensemble $STrans$ n'est pas mis à jour car aucun nouveau symbole n'a été inséré.

Algorithme 9.2.3 - Construction de $RTStates$ et $STrans$

```
01. Input: M_E = (Σ, Q, q_0, F, Δ), G = (X, U), et TMaps
02. Output: RTStates et STrans
03. s_nl = q_0, sub_w = ε
04. for each row row of TMaps do {
05. // construit un sous-mot du mot d'origine à partir du dernier état valide
06.   sub_w = sub_w · TMaps[row, old] // du mot mis à jour.
07.   if (TMaps[row, old] = ε) {
08.     insertionSubString(s_nl, row, TMaps, M_E, sub_w, RTStates, STrans)
09.   }
10.   else {
11.     if (TMaps[row, new] = ε){
12.       deletionSubString(s_nl, row, TMaps, M_E, sub_w, RTStates)
13.     }
14.     else {
15.       s_nl = δ̂(s_nl, TMaps[row, old])
16.       sub_w = ε // sub_w aura les symboles valides à partir du dernier s_nl trouvé.
17.     }
18.   }
19. } // end for
```

L'algorithme 9.2.3 cherche à construire des triplets $\langle s_{nl}, s_{nr}, s_{new} \rangle$ pour chaque ligne dans $TMaps$ où l'une des colonnes est ε. Ainsi, l'avantage d'utiliser $TMaps$ pour guider la recherche des triplets est de réduire l'espace de recherche des états s_{nl} et s_{nr} dans le mot mis à jour.

Rappelons que la recherche de l'état s_{nr} dans le cadre de mise à jour unique est directe : en se servant du mot d'origine et de la position de mise à jour, il est trouvé. Étant donné le mot $w[0 : n]$, la position de mise à jour p $(p < n)$ et l'état s_{nl}, $s_{nr} = \hat{\delta}(s_{nl}, w[p])$. Néanmoins, dans le cadre de mise à jour multiples, ce raisonnement peut ne plus être valable. Considérons le mot $w = abcd$ qui appartient au langage décrit par $E = abcd$ ($\overline{E} = 1\ 2\ 3\ 4$). Supposons deux

opérations de mises à jour sur w : $(1, delete, b)$ et $(2, insert, x)$, qui résulte le mot $w' = axcd$. L'état s_{nl} est égal à 1 avant que le symbole x soit lu (*i.e.*, $s_{nl} = \delta(0, a)$). Si on utilise le même principe de GREC pour trouver s_{nr}, alors le calcul est $s_{nr} = \delta(1, c)$. Il est clair que, dans ce cas, $s_{nr} = \perp$ car la transition $\delta(1, c)$ n'existe pas dans l'automate de E. En effet, dans le cadre de GREC-e, une même suite de mise à jour peut avoir de suppressions et de insertions et si des symboles obligatoires sont supprimés du mot d'origine, alors il n'existera plus une liaison valide entre le dernier état s_{nl} et la position d'une insertion, comme le montre l'exemple ci-dessus.

Pour résoudre cela, nous utilisons un sous-mot de w qui représente une séquence de symboles dont un état valide est trouvé à partir de s_{nl}. Ainsi, en utilisant ce sous-mot sub_w, on peut écrire $\hat{\delta} = (s_{nl}, sub_w)$ et l'état résultat sera toujours différent de $\perp$. L'utilisation de sub_w est une optimisation pour trouver s_{nr} car on pouvait utiliser une autre approche plus coûteux : utiliser le préfixe $w[0 : p]$ (où p est la position de mise à jour). Ainsi, pour chaque position p_i d'une opération de mise à jour, s_{nr} serait trouvé comme suit : $s_{nr} = \hat{\delta}(0, w[0 : p_i])$. Remarquer que l'automate doit être parcouru dès le début pour chaque s_{nr} à traiter. En utilisant sub_w, s_{nr} peut être trouvé en parcourant l'automate à partir de s_{nl} avec le sous-mot sub_w et $|sub_w| \leq |w[0 : p_i]|$ pour une position de mise à jour p_i quelconque.

Avant de considérer les procédures *insertionSubString* et *deletionSubString* de l'algorithme 9.2.3, nous analysons la fonction *NextValidSymbol* (figure 9.1). Cette fonction représente la partie la plus importante dans la construction de la relation $RTStates$. Les paramètres d'entrée sont les suivants :

1. Un état qui représente l'état s_{nl}.
2. Une chaîne de caractères β qui, initialement, correspond à $TMaps[row, new]$ pour une ligne row donnée.
3. Une position j dans β.
4. $TMaps$.
5. Un entier l qui indique une ligne de $TMaps$.
6. L'automate d'origine M_E.
7. Le sous-mot sub_w utilisé, si nécessaire, pour guider la recherche de l'état s_{nr}.

La fonction *NextValidSymbol* cherche, en utilisant les symboles présents dans les lignes de la colonne new de $TMaps$ à partir de la ligne l, un symbole *symbol* qui étiquette une transition depuis s_{nl}. L'état cible de cette transition sera s_{nr}. Remarquer que s_{nl} représente le dernier état valide de M_E trouvé en lisant les symboles de $TMaps[row, old]$, pour $row < l$.

Les suppressions et les insertions ont un traitement différent dans l'initialisation de la fonction :

1. Dans le cadre de la suppression, β est initialement vide (ε) et j est égal à -1 car, comme $TMaps[l, new]$ est vide, il ne contient pas de symbole pour trouver s_{nr}.
2. Dans le cadre de l'insertion, le β représente le sous-mot $TMaps[l, new]$ et j est la position dans β qui correspond à un symbole du nouveau mot.

Dans le deux cas, le sous-mot β guide la recherche de l'état s_{nr}.

L'approche utilisée dans la fonction *nextValidSymbol* peut être résumée comme suit : l'état s_{aux} est celui trouvé dans M_E après la lecture de $sub_w = w[i : k - 1]$, pour une position k de mise à jour ($i < k$). Remarquer que $s_{nl} = \hat{\delta}(0, w[0 : i])$. Pour trouver le prochain symbole qui étiquette une transition valide de M_E, *nextValidSymbol* effectue les vérifications suivantes :

- S'il existe une transition valide à partir de s_{aux} étiqueté $\beta[j]$, alors *nextValidSymbol* retourne la position j et l'état s_{aux}.
- Si la transition $\delta(s_{aux}, \beta[j])$ n'est pas valide, *nextValidSymbol* prend le prochain symbole du mot β pour chercher une transition valide.

```
function nextValidSymbol(s_nl, β, j, TMaps, l, M_E, sub_w) {
//La fonction cherche un symbole qui correspond à une étiquette d'une transition
//de M_E à partir de s_nl
01. isLimit = false, α = ε
02. // s_aux est l'état valide par rapport au mot d'origine (avant la position de maj)
03. s_aux = δ̂(s_nl, sub_w)
04. length = |β|
05. if (β = ε) { symbol = ∅ } else { symbol = β[j] }
07. while (¬isLimit and δ(s_aux, symbol) =⊥) do {
08.   if (j = (length − 1)) { //Une nouvelle ligne de TMaps doit être analysée
10.     l = l + 1                    //car le dernier symbole de β a été lu.
09.     if (TMaps[l, old] ≠ ε and TMaps[l, new] ≠ ε) {
11.     //Un sous-mot de w non-changé est trouvé
12.     //Le premier symbole de TMaps[l, new] est concaténé à β
13.       β = β · TMaps[l, new][0]
14.       α = α · TMaps[l, new][0]
15.       // La recherche doit finir car une ligne avec des symboles non-changés a été trouvée.
16.       isLimit = true
17.     } else {
18.       // Les prochains symboles de TMaps sont concaténés à β (s'il existent)
19.       // et la recherche pour un état valide pour s_nr doit continuer.
20.       β = β · TMaps[l, new]
21.       α = α · TMaps[l, old]
22.     }
23.     length = |β|
24.   }
25.   if (TMaps[l, new])≠ ε)
26.     j = j + 1          // j doit être incrémenté, sauf s'il s'agit d'une suppression.
27.     symbol = β[j]
28.   }
29. }
30. // Par définition il existe toujours une transition valide de s_nl à
31. // un autre état en utilisant les informations dans TMaps.
32. // Si un état n'est pas trouvé le sous-mot α est utilisé pour le trouver.
33. if (δ(s_aux, symbol) =⊥) {
34.   s_aux = δ̂(s_aux, α)
35.   symbol = ε // deletionSubString doit construire un triplet
36. }
37. return (j, s_aux) }
```

FIG. 9.1 – La fonction *nextValidSymbol*

Remarquer que β est construit par la concaténation des mots dans $TMaps[l, new]$. Cette concaténation est faite au fur et à mesure que nous avançons sur β sans trouver une transition valide (*i.e.*, au fur et à mesure que j est incrémenté). La recherche sur β s'arrête (i) lorsqu'une transition $\delta(s_{aux}, \beta[j])$ valide est trouvée ou (ii) lorsque $TMaps[l, new]$ représente un sous-mot qui n'a pas été modifié par les mises à jour. Le deuxième cas indique qu'un symbole non mis à jour a été lu (il existe donc dans w et w'). Si, en utilisant le symbole non modifié, une transition $\delta(s_{aux}, \beta[j])$ valide n'est pas trouvée, alors il y a eu des suppressions des symboles obligatoires dans le sous-mot β considéré. Dans ce cas, il faut utiliser le mot α pour la recherche d'une transition valide. Le sous-mot α est construit de façon similaire à celle utilisée pour obtenir β. Néanmoins, α est construit à partir du mot d'origine w.

Les procédures *insertionSubString* et *deletionSubString* traitent le retour de *nextValidSymbol* de la manière suivante :

- *insertionSubString* (figure 9.2) : si la position retournée est différente de la position donnée comme paramètre, alors il est nécessaire d'insérer un triplet dans $RTStates$ car le symbole dans la position donnée n'a pas une transition valide à partir de s_{nl} dans l'automate d'états finis.
- *deletionSubString* (figure 9.3) : si la position retournée est égal à -1, alors il est nécessaire d'insérer un triplet dans $RTStates$ car aucun symbole après le sous-mot supprimé n'a une transition valide à partir de s_{nl}.

```
procedure insertionSubString(s_nl, row, TMaps, M_E, sub_w, RTState, STrans) {
// Elle vérifie s'il est nécessaire d'insérer un nouveau triplet dans RTState
// lorsque des nouveaux symboles sont insérés dans le mot d'origine.
// La séquence d'insertion est considérée comme l'insertion d'un sous-mot.
01. α = TMaps[row, new]
02. s_new = null
03. for i from 0 to |α| − 1
04. // Tous les symboles insérés sont traités, i.e., il est vérifié
05. // si un nouvel état doit être créé et aussi les transitions de cet état
06. // dans l'automate d'états finis M_E
07.   LastNew = s_new
08.   (pos, foundState) = nextValidSymbol(s_nl, α, i, TMaps, row, M_E, sub_w)
09.   s_nr = foundState
10.   if (pos > i) { // α[i] is an invalid symbol
11.     let s_new be a new state built for α[i]
12.     if ({ ∄⟨s'_nl, s'_nr, s'_new⟩ in RTStates | s_nl = s'_nl, s'_nr = s_nr, f(s_new) = f(s'_new)}) {
13.       insert ⟨s_nl, s_nr, s_new⟩ into RTStates
14.     } else { STrans = STrans ∪ {(s_nl, α[i])}}
15.     if (i > 0 and δ(s_nl, α[i − 1]) =⊥) {
16.         STrans = STrans ∪ {(LastNew, α[i])}
17.     }
18.   }
19.   else s_nl = s_nr
20. } } // end for i and end procedure
```

FIG. 9.2 – Procédure *insertionSubString*

Remarquons que la procédure *insertionSubString* (figure 9.2) cherche à ne pas insérer deux triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ et $\langle s'_{nl}, s'_{nr}, s'_{new}\rangle$ tels que $s_{nl} = s'_{nl}$, $s_{nr} = s'_{nr}$ et s_{new} et s'_{new} représentent le même symbole dans le mot mis à jour. En imposant cette restriction, on évite d'ajouter des symboles répétés dans une même séquence.

Par exemple, dans la procédure *insertionSubString*, étant donné $w = abc$, $w' = abxxc$ et $E = abc$ ($\overline{E} = 1\ 2\ 3$), si le test de la ligne 12 n'était pas fait, on aurait les triplets $\langle 2, 3, 4\rangle$ et $\langle 2, 3, 5\rangle$ dans $RTStates$, alors que les états 4 et 5 représentant le même symbole (*i.e.*, x).

La procédure *insertionSubString* est responsable aussi de construire l'ensemble $STrans$. Pour chaque triplet inséré dans $RTStates$, *insertionSubString* cherche à construire aussi un couple (e, a) de l'ensemble $STrans$. Cela s'explique car les triplets $TStates$ sont construits indépendamment les uns des autres et il faut garantir que :

- S'il existe deux ou plusieurs symboles différents non reconnus de suite dans le nouveau mot, alors les automates qui représentent les expressions régulières candidates (leurs automates) doivent avoir des transitions entre ces symboles (condition dans la ligne 15).
- L'insertion de symboles à répétition soit traitée en éliminant les triplets doublons (voir l'exemple 9.2.5), mais en assurant que les nouveaux automates ont une boucle pour représenter cette répétition. Une nouvelle paire est insérée dans $STrans$ (ligne 14).

L'exemple ci-dessous montre l'utilisation de $STrans$ dans les deux cas mentionnés ci-dessus.

Exemple 9.2.5 Soient $E = abc$ ($\overline{E} = 1\ 2\ 3$) une expression régulière et $w = abc$ un mot qui appartient à $L(E)$. Soit $w' = ab\texttt{xxy}c$ le mot w mis à jour avec l'insertion des symboles x, x et y. Soient 4 l'état qui représente x et 5 l'état qui représente y. Dans ce contexte, $RTStates = (\langle 2, 3, 4\rangle, \langle 2, 3, 5\rangle)$ et l'ensemble $STrans$ a les couples suivants :

- $(4, y)$ qui garantit que les automates construits à partir des expressions régulières proposées ont des transitions depuis l'état 4 (qui représente le symbole x) étiquetées y, c'est-à-dire, les mots qui appartient aux langages des expressions régulières candidates peuvent avoir des sous-mots composés par xy.
- $(4, x)$ qui garantit que les automates construits à partir des expressions régulières proposées ont des transitions depuis l'état 4 (qui représente le symbole x) étiquetée x, c'est-à-dire, x doit être décoré avec une *. □

La procédure *deletionSubString* (figure 9.3), en revanche, ne cherche pas à trouver pour chaque symbole supprimé dans $TMaps[row, old]$ un triplet $\langle s_{nl}, s_{nr}, null\rangle$. En fait, elle vérifie s'il n'existe pas une transition entre s_{nl} et les prochains symboles qui existent dans le nouveau mot. Si une transition est trouvée, alors le sous-mots supprimé ne représente pas des symboles obligatoires et la valeur de la position retournée est supérieure ou égale à 0, sinon la fonction *nextValidSymbol* retourne la valeur -1 pour *pos*. Cette condition est testée dans la ligne 02, si la condition est vérifiée alors nous insérons[3] dans la relation $RTStates$ un nouveau triplet (ligne 06).

```
procedure deletionSubString(s_nl, row, TMaps, M_E, sub_w) {
// A new row in TMaps should be investigated
01. (pos, foundState)= nextValidSymbol(s_nl, ε, −1, TMaps, row, M_E, sub_w)
02. if (pos < 0) {
03.    //TMaps has been verified until finding a substring of the old word w.
04.    //No valid transition from s_nl labeled symbol was found.
05.    s_nr = foundState
06.    insert ⟨s_nl, s_nr, null⟩ into RTStates
07. } }
```

FIG. 9.3 – Procédure *deletionSubString*

[3]Les doublons ne sont pas insérés.

L'exemple 9.2.6 présente les résultats de l'exécution des algorithmes présentés ci-dessus.

Exemple 9.2.6 Soient $E = q_{Sujet}\ (q_{Annee}\ {q_{Revue}}^{+})^{*}\#\ (\overline{E} = 1\ (2\ 3^{+})^{*}4)$ une expression régulière, $w = q_{Sujet}\ q_{Annee}\ q_{Revue}\ q_{Revue}\ q_{Annee}\ q_{Revue}\ q_{Annee}\ q_{Revue}\ q_{Revue}\#$ un mot tel que $w \in L(E)$ et

1. $(4, insert, q_{Conference})$.
2. $(4, insert, q_{Rapport})$.
3. $(8, delete, \emptyset)$.
4. $(9, insert, q_{Conference})$.
5. $(9, insert, q_{Conference})$.
6. $(9, insert, q_{Annee})$.
7. $(9, insert, q_{Revue})$.
8. $(9, insert, q_{Revue})$,

la relation *UpdateTable* appliqueé sur w. Le tableau $TMaps$ construit par l'algorithme 9.2.1 est :

old	*new*
$q_{Sujet}\ q_{Annee}\ q_{Revue}\ q_{Revue}$	$q_{Sujet}\ q_{Annee}\ q_{Revue}\ q_{Revue}$
ε	$q_{Conference}\ q_{Rapport}$
$q_{Annee}\ q_{Revue}\ q_{Annee}\ q_{Revue}$	$q_{Annee}\ q_{Revue}\ q_{Annee}\ q_{Revue}$
q_{Revue}	ε
ε	$q_{Conference}\ q_{Conference}\ q_{Annee}\ q_{Revue}\ q_{Revue}$
$\#$	$\#$

La relation $RTStates = (\langle 3, 2, 5\rangle, \langle 3, 2, 6\rangle)$, où 5 est l'état qui représente $q_{Conference}$ et 6 représente $q_{Rapport}$. L'ensemble $STrans = \{(5, q_{Rapport}), (5, q_{Conference})\}$ ce qui veut dire que les nouveaux automates construits à partir de candidates générées par `GREC-e` doivent avoir les transitions $\delta(5, q_{Rapport}) \neq \perp$ et $\delta(5, q_{Conference}) \neq \perp$ dans leurs ensembles de transitions. □

Pour terminer cette section nous allons présenter brièvement la complexité des algorithmes présentés. La construction du tableau $TMaps$ est linéaire par rapport au nombre de symboles du mot d'origine $|w|$ plus le nombre de triplets dans *UpdateTable*, noté $|UpdateTable|$, c'est-a-dire, $O(|w| + |UpdateTable|)$.

Pour la construction de la relation $RTStates$ et $STrans$, l'algorithme 9.2.3 parcourt toutes les lignes de $TMaps$ (n_l). Pour chaque sous-mot w_s dans des lignes dont la colonne *new* ou *old* est vide, la procédure *insertionSubString* parcourt w_s du début à la fin (p_i) et la procédure *deletionSubString* traite w_s d'un seul coup. Pour *insertionSubString* et *deletionSubString*, la fonction *nextValidSymbol* parcourt $TMaps$ à partir de la ligne donnée comme paramètre jusqu'à la prochaine ligne dont les colonnes *old* et *new* sont pas vides (p_l). Le nombre de symboles concaténés (c_s) pendant le parcours représente le nombre de fois que la boucle *while* est exécutée. Pour chaque répétition de la boucle *while*, une règle de transition de l'automate est testée, alors le nombre de règles à tester est égal à c_s. Un même symbole s du mot mis à jour w' peut être lu n fois (où n représente la position de s par rapport au début du mot) pour trouver l'état s_{nr} pour un symbole s' dans la position n' de w' telle que $n' < n$. Ainsi, si on considère le cas le pire, c'est-à-dire, tous les symboles du mot d'origine ont été supprimés et des nouveaux symboles ont été insérés, alors $TMaps$ aura 3 lignes, l'une avec les symboles supprimés, l'autre avec les symboles insérés et la troisième avec le symbole # dans les deux colonnes. Étant donné le sous-mot supprimé sw_s et le sous-mot inséré sw_i, la recherche de s_{nr} pour la suppression est faite en parcourant les symboles dans sw_i jusqu'au premier symbole dans la troisième ligne, la recherche des états s_{nr} pour les symboles dans sw_i est faite en parcourant $\frac{|sw_i|(|sw_i|-1)}{2}$ fois le

```
   Function GREC-e (G_wo, H, RTStates, STrans) {
01.  setRegExp = ∅
02.  if (RTStates est vide) {
03.     return (setRegExp = setRegExp ∪ {GraphToRegExp(G_wo, H)})
04.  }
05.  if (G_wo n'a qu'un nœud) { return ∅ }
06.  R_i :=ChooseRule(G_wo, H)
07.  for each (G_new, H_new, RTStates') :=
                LookForGraphAlternative-e (G_wo, H, R_i ,RTStates, STrans) do
08.     setRegExp = setRegExp ∪ GREC-e(G_new, H_new, RTStates', STrans)
09.  (G'_wo,H') := ApplyRule(R_i, G_wo,H)
10.  setRegExp = setRegExp ∪ GREC-e(G'_wo, H', RTStates, STrans)
   return setRegExp }
```

FIG. 9.4 – `GREC-e` : version étendue de `GREC`.

sous-mot sw_i et les règles de transitions de l'automate d'états finis sont vérifiées le même nombre de fois. Alors, on conclut que la complexité en temps d'exécution de la construction de $RTStates$ et $STrans$ est en

$$O(n_l + p_i + \frac{s_i(s_i-1)}{2} + \frac{s_i(s_i-1)}{2}),$$

où s_i est le nombre total des symboles insérés et supprimés du mot d'origine.

Ainsi, la complexité de la construction de la relation $RTStates$ est linéaire sur la taille du mot et quadratique sur le nombre de symboles insérés et supprimés.

Avec la relation $RTStates$ et l'ensemble $STrans$, `GREC-e` peut construire des candidats pour remplacer une expression régulière d'origine dans une règle de transition de l'automate d'arbre qui représente le schéma XML. La section suivante présente cette approche.

9.3 L'algorithme `GREC-e`

`GREC-e` utilise l'algorithme proposé dans [CZ00] qui réduit un graphe de Glushkov en l'expression régulière correspondante pour guider la recherche des expressions régulières candidates. Néanmoins, `GREC-e` traite plusieurs triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$. La figure 9.4 montre l'algorithme `GREC-e`.

Le problème de proposer des candidats pour remplacer un schéma XML d'origine peut être décrit comme suit : étant donné une expression régulière E, un mot w tel que $w \in L(E)$ et un mot w' résultat des mises à jour sur w, proposer de nouvelles expressions régulières E' telles que $L(E) \cup \{w'\} \subseteq L(E')$ et $\mathcal{D}(E, E') = n$ (où n est le nombre de triplets dans $RTStates$ dont s_{new} est différent de $null$).

La sémantique des états s_{nl} et s_{nr} n'a pas changé par rapport à celle de l'approche `GREC` : s_{nl} représente l'état le plus proche à gauche du symbole non-reconnu et s_{nr} représente l'état le plus proche à droite du symbole non-reconnu. Pour les insertions, s_{new} représente toujours le nouveau nœud à insérer dans le graphe qui représente l'automate de Glushkov. Pour les suppressions, s_{new} n'est plus utilisé, `LookForGraphAlternative-e` utilise les nœuds représentés par s_{nl} et s_{nr} pour rendre optionnel les nœuds obligatoires entre ces deux nœuds.

Étant donné un graphe de Glushkov sans orbite G_{wo} et sa hiérarchie d'orbites $\mathcal{H}$, la vérification de la possibilité d'insertion des nouveaux nœuds à chaque pas du processus de réduction du graphe (de l'automate) d'origine est faite comme suit :

1. On applique `GREC-e` sur G_{wo}, $\mathcal{H}$, $RTStates$ et $STrans$: avant d'appliquer une des règles de réduction sur G_{wo}, on exécute des tests qui sont différents pour chaque règles $\mathbf{R}_1$, $\mathbf{R}_2$ ou $\mathbf{R}_3$.
2. Avant l'application d'une règle R_i dans le processus de réduction, on teste si les nœuds s_{nl} et s_{nr} dans des triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ de $RTStates$ sont affectés par R_i. Cela est fait dans la fonction `LookForGraphAlternative-e`.
   (a) Tant qu'il existe des triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ tels que s_{nl} et s_{nr} sont affectés par R_i alors : si $s_{new} \neq null$, le nœud s_{new} est inséré dans le graphe, sinon des arcs de s_{nl} vers s_{nr} sont ajoutés au graphe. Cette procédure génère un nouveau graphe G_{new} et une nouvelle relation $RTStates'$ résultante de la suppression des triplets utilisés par `LookForGraphAlternative-e`. Un nouvel appel à `GREC-e` est fait :
      i. Une nouvelle instance de `GREC-e` recommence et les pas 1 et 2 sont répétés.
      ii. Lorsque la relation $RTStates$ est vide, l'expression régulière correspondant à toutes les mises à jour est construite et l'instance de `GREC-e` termine.
   (b) Sinon, aucune modification n'est exécutée sur le graphe.
3. `GREC-e` applique R_i sur G_{wo} et le processus continue comme expliqué dans les items 1 et 2. Le processus s'arrête lorsque G_{wo} a seulement un nœud.
4. L'ordre d'application des règles $\mathbf{R}_1$, $\mathbf{R}_2$ et $\mathbf{R}_3$ ainsi que les nœuds sur lesquels les appliquer sont donnés par $\mathcal{H}$.

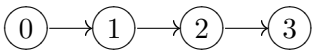
Notons que les pas présentés par les items $2(a)$ et $2(b)$ ci-dessus sont exécutés par la fonction `LookForGraphAlternative-e` qui est la responsable de la construction des nouveaux graphes candidats qui serviront de paramètres à un nouvel appel de `GREC-e`.

9.3.1 La fonction `LookForGraphAlternative-e` : génération des graphes candidats

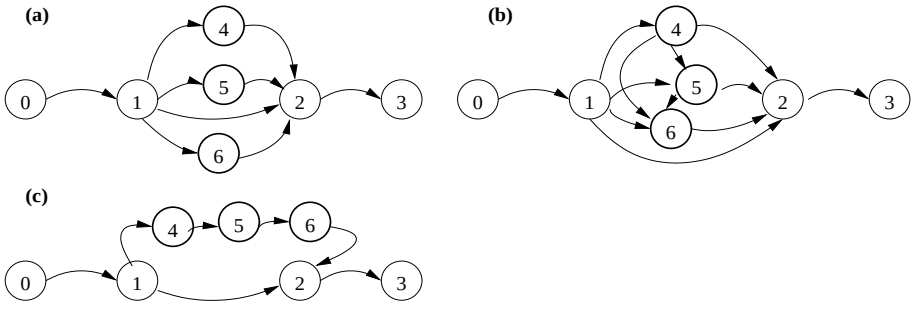
Le rôle de la fonction `LookForGraphAlternative-e` est essentiel puisque c'est dans cette fonction que sont générés les graphes qui seront transformés en candidats proposés par `GREC-e`. Les modifications apportées au graphe à réduire sont guidées par les règles de réduction et par la hiérarchie des orbites du graphe.

Toutes les définitions utilisées par `LookForGraphAlternative` (définitions 8.4.3 à 8.4.6) de `GREC` pour construire les graphes candidates sont utilisées par `LookForGraphAlternative-e`. Cependant, `LookForGraphAlternative-e` peut insérer en même temps plusieurs nœuds s_{new} et, lorsque les nœuds s_{new} peuvent être groupés, on peut avoir plusieurs manières de les grouper.

Exemple 9.3.1 Soit G le graphe ci-dessous :



Soient 4, 5 et 6 des nœuds à insérer entre les nœuds 1 et 2 de manière telle qu'ils représentent des chemins optionnels entre les nœuds 1 et 2. La figure ci-dessous montre des possibilités d'insertions des nœuds 4, 5 et 6 dans G.



Remarquer que l'on considère un ordre entre les nœuds à insérer. Cet ordre peut être l'ordre dans lequel les symboles représentés par les nœuds apparaissent dans le mot considéré. Le graphe **(a)** correspond à l'expression régulière 1 2 (4|5|6)! 3, le graphe **(b)** correspond à l'expression régulière 1 2 4! 5! 6! 3 et le graphe **(c)** correspond à l'expression régulière 1 2 (4 5 6)! 3. □

Ainsi, `LookForGraphAlternative-e` utilise des extensions des définitions 8.4.3 à 8.4.6 pour traiter le contexte présenté dans l'exemple 9.3.1. `LookForGraphAlternative-e` a les caractéristiques suivantes :

1. Les nouvelles définitions doivent prendre en considération que plus d'un triplet $\langle s_{nl}, s_{nr}, s_{new}\rangle$ peuvent vérifier les conditions d'une définition. Dans ce cas, plus d'un nœud s_{new} sont insérés dans le graphe d'origine à la fois. On peut grouper ces nœuds de plusieurs manières et `LookForGraphAlternative` utilise trois manières différentes :
   - Les nouveaux nœuds sont groupés de façon disjointe entre s_{nl} et s_{nr}, comme le graphe **(a)** de l'exemple 9.3.1
   - Les nouveaux nœuds sont groupés de façon optionnel entre s_{nl} et s_{nr}, comme le graphe **(b)** de l'exemple 9.3.1
   - Les nouveaux nœuds sont groupés de façon séquentielle entre s_{nl} et s_{nr}, comme le graphe **(c)** de l'exemple 9.3.1
2. Les triplets de $RTStates$ sont supprimés après qu'ils aient été traités.
3. Avant de proposer un nouveau graphe G_{new}, `LookForGraphAlternative` teste si les arcs de G_{new} respectent les transitions dans $STrans$.

Dans la suite, nous présentons l'extension de la règle $\mathbf{R}_1$ utilisée par `GREC-e`. Cette extension est illustrée par la figure 9.5. Nous ne présentons pas les extensions pour les autres règles car le principe est similaire.

Définition 9.3.1 Candidats générés à partir de $\mathbf{R}_1$ par `LookForGraphAlternative-e` : Soit $G_{wo} = (X, U)$ un graphe de Glushkov et $x, y \in X$ deux nœuds sur lesquels $\mathbf{R}_1$ peut être appliquée. Soit ν une fonction qui fait correspondre un symbole e au nœud correspondant dans un graphe. Soit n_{new} un nœud sans expression régulière associée. Les graphes $G^i = (X^i, U^i)$ qui peuvent être proposés à partir de G_1 sont définis ci-dessous. Soient $x^i = x$ et $y^i = y$ tel que $x^i, y^i \in X^i$.

Cas 1 :

Initialiser $\mathbf{G}^1$ et $\mathbf{G}^2$ avec (X, U)

`pour chaque triplet` $\langle s_{nl}, s_{nr}, s_{new}\rangle$ `dans` $RTStates$ `tel que` $s_{nl} = x$ `et` $s_{nr} = y$ `faire`

$\mathbf{G}^1$: $X^1 = X^1 \cup \{s_{new}\}$; $U^1 = U^1 \cup \{(x^1, s_{new})\} \cup \{(s_{new}, y^1)\}$.

$\mathbf{G}^2$: $X^2 = X^2 \cup \{s_{new}\}$; $U^2 = U^2 \cup \{(x^2, s_{new})\} \cup \{(s_{new}, y^2)\} \cup \{(z, s_{new}) \mid z \in Q^+(x^2) \setminus \{y^2\}\}$.

$n_{new} = n_{new} \cdot s_{new}$

supprimer $\langle s_{nl}, s_{nr}, s_{new}\rangle$ de $RTStates$

`fin pour.`

$\mathbf{G}^3$: $X^3 = X \cup \{n_{new}\}$; $U^3 = U \cup \{(x, n_{new})\} \cup \{(n_{new}, y)\}$.

`pour chaque graphe` G^i `faire`

`si` $(\exists(a, \nu(e)) \in STrans | \forall(a', e') \in U^i$, a'$\neq a$ et $e' \neq \nu(e))$ `alors` abandonner G^i

`fin pour`

Cas 2 :

Initialiser $\mathbf{G}^1$, $\mathbf{G}^2$, $\mathbf{G}^3$ et $\mathbf{G}^4$ avec (X, U)

`pour chaque triplet` $\langle s_{nl}, s_{nr}, s_{new}\rangle$ `dans` $RTStates$ `tel que` $s_{nl} = y$ `et` $s_{nr} = x$ `faire`

$\mathbf{G}^1$: $X^1 = X^1 \cup \{s_{new}\}$; $U^1 = U^1 \cup \{(y^1, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(y)\}$.

$\mathbf{G}^2$: $X^2 = X^2 \cup \{s_{new}\}$; $U^2 = U^2 \cup \{(s_{new}, x^2)\} \cup \{(z, s_{new}) \mid z \in Q^-(x)\}$.

$\mathbf{G}^3$: $X^3 = X^3 \cup \{s_{new}\}$; $U^3 = U^3 \cup \{(s_{new}, x^3)\} \cup \{(z, s_{new}) \mid z \in Q^-(x^3)\}$.

$\mathbf{G}^4$: $X^4 = X^4 \cup \{s_{new}\}$; $U^4 = U^4 \cup \{(y^4, s_{new})\} \cup \{(s_{new}, z) \mid z \in Q^+(y)\} \cup \{(v, s_{new}) \mid v \in Q^+(y^1) \setminus Q^+(y)\}$.

$n_{new} = n_{new} \cdot s_{new}$
supprimer $\langle s_{nl}, s_{nr}, s_{new}\rangle$ de $RTStates$
`fin pour.`
$\mathbf{G}^5$: $X^5 = X \cup \{n_{new}\}$; $U^5 = U \cup \{(y, n_{new})\} \cup \{(n_{new}, z) \mid z \in Q^+(y)\}$.
$\mathbf{G}^6$: $X^6 = X \cup \{n_{new}\}$; $U^6 = U \cup \{(n_{new}, x)\} \cup \{(z, n_{new}) \mid z \in Q^-(x)\}$.
`pour chaque graphe` G^i `faire`
`si` $(\exists (a, \nu(e)) \in STrans \mid \forall (a', e') \in U^i$, a'$\neq a$ et $e' \neq \nu(e))$ `alors` abandonner G^i
`fin pour` □

Les conditions de la définition 9.3.1 sont les mêmes que celles de la définition 8.4.3, ce qui change est le traitement de plusieurs triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ à la fois, par une boucle *pour chaque*. Cette boucle construit trois nouveaux graphes pour chaque graphe qu'aurait construit la définition 8.4.3. Remarquer que si le nombre de triplets à traiter par la définition 9.3.1 est égal à 1, la définition 9.3.1 propose le même nombre de graphes que la définition 8.4.3.

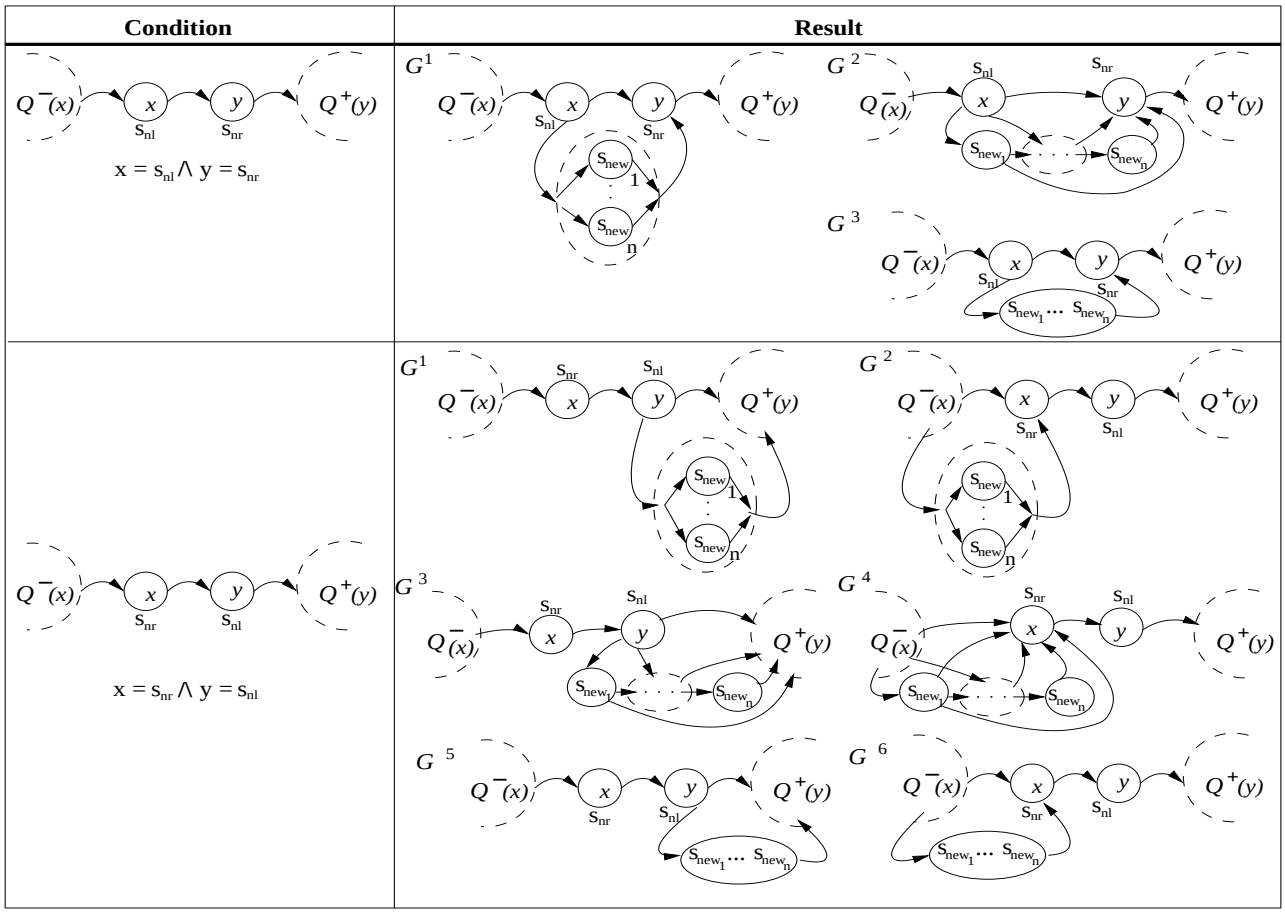
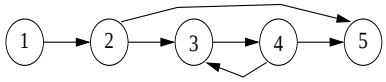


FIG. 9.5 – Conditions et modifications testées par `LookForGraphAlternative-e` avant d'appliquer la règle $\mathbf{R}_1$.

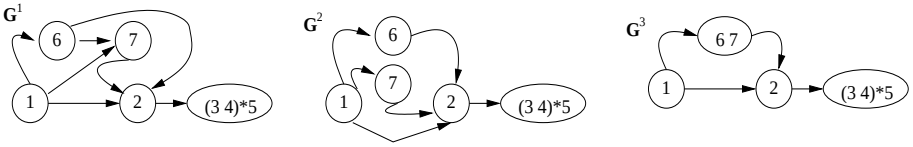
L'exemple 9.3.2, ci-dessous, illustre l'utilisation des deux cas de la définition 9.3.1.

Exemple 9.3.2 Soient $E = ab(cd)^*e$ ($\overline{E} = 1\ 2(3\ 4)^*5$) une expression régulière, $w = abcde$ un mot tel que $w \in L(E)$ et $w' = axybcde$ le mot w mis à jour avec l'insertion de x et y. Soit G le graphe de Glushkov[4] construit à partir de E :

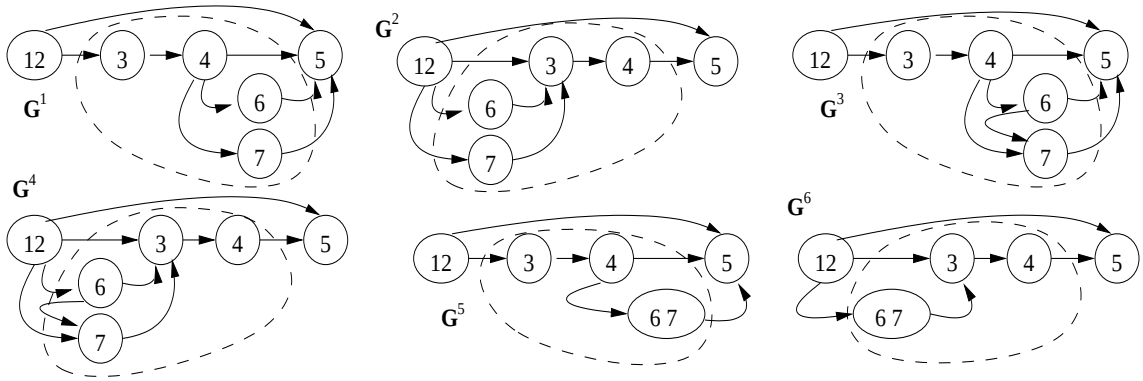
[4]Pour simplifier les figures, nous avons omis le nœud 0.



Soient 6 et 7 les nœuds associés aux symboles x et y, respectivement. Soient $\langle 1, 2, 6\rangle$ et $\langle 1, 2, 7\rangle$ les triplets dans $RTStates$ calculés par l'algorithme 9.2.3. Soit $(6, y)$ le couple dans $STrans$. Supposons que la règle $\mathbf{R}_1$ est appliquée sur le graphe G construit à partir de E sur les nœuds 1 e 2 et que les nœuds 3, 4 et 5 ont été déjà fusionnés.
À ce moment, `LookForGraphAlternative-e` applique la définition 9.3.1, cas 1 et les graphes construits sont :



Les expressions régulières construites à partir des graphes G^1, G^2 et G^3 sont $ax!y!b(cd)^*e$, $a(x|y)^*b(cd)^*e$ et $a(xy)!b(cd)^*e$, respectivement. Remarquer que la solution $a(x|y)?b(cd)^*e$ n'est pas proposée car y n'y est pas successeur de x comme l'oblige le couple $(6, y)$ de $STrans$.
Supposons, maintenant, que $w = abcdcde$ et $w' = abcdxycde$. Dans ce contexte, $RTStates = (\langle 4, 3, 6\rangle, \langle 4, 3, 7\rangle)$ et $STrans = \{(6, y)\}$. Supposons aussi que la règle $\mathbf{R}_1$ est appliquée sur les nœuds 3 e 4 et que les nœuds 1, 2 ont été déjà fusionnés.
À ce moment, `LookForGraphAlternative-e` applique la définition 9.3.1, cas 2 et les graphes construits sont :



Six nouveaux graphes (G^1, G^2, G^3, G^4, G^5 et G^6) sont donc proposés à partir du graphe d'origine. Les expressions régulières construites correspondant à G^1, G^2, G^3, G^4, G^5 et G^6 sont $ab(cd(x|y)^*)^*e$, $ab((x|y)^*cd)^*e$, $ab(cdx!y!)^*e$, $ab(x!y!cd)^*e$, $ab(cd(xy)!)^*e$ et $ab((xy)!cd)^*e$, respectivement. Remarquer encore que les expressions régulières $ab(cd(x|y)?))^*e$ et $ab((x|y)?cd)^*e$ ne sont pas proposées car leurs automates n'ont pas une transition $\delta(6, y) \neq \perp$. □

Dans la suite nous montrons comment `LookForGraphAlternative-e` traite les suppressions.

Suppressions invalides

Si la relation $UpdateTable$ contient des opérations de mises à jour du type *delete* et/ou *replace* et si ces opérations affectent des positions qui représentent des symboles obligatoires dans un mot w, le mot résultant w' n'appartiendra plus au langage auquel w appartient. Les suppressions de symboles obligatoires d'un mot sont représentées dans $RTStates$ par des triplets de la forme $\langle s_{nl}, s_{nr}, null\rangle$, où s_{nl} et s_{nr} sont les nœuds qui doivent guider `LookForGraphAlternative-e` pour proposer un graphe selon lequel les symboles obligatoires supprimés deviennent des symboles optionnels. La définition 8.4.7 utilisée par `LookForGraphAlternative` pour rendre optionnel le symbole obligatoire supprimé n'est plus valide dans le contexte de mises à jour multiples car, auparavant, s_{new} représentait le symbole qu'il fallait rendre optionnel et maintenant, c'est

s_{nl} et s_{nr} qui délimitent l'ensemble des symboles qu'il faut rendre optionnels. La définition ci-dessous formalise cette idée.

Définition 9.3.2 Soit E une expression régulière. Soit $G_1 = (X_1, U_1)$ un graphe de Glushkov qui représente l'automate construit de E. Soit $RTStates$ la relation qui stocke des triplets $\langle s_{nl}, s_{nr}, null\rangle$. Nous définissons des nouveaux graphes $G^i = (X^i, U^i)$ de G_1 comme suit:

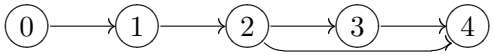
`pour chaque` $\langle s_{nl}, s_{nr}, null\rangle$ `dans` $RTStates$ `faire`
  $\mathbf{G}^1 : X^1 = X_1; U^1 = U_1 \cup \{(s_{nl}, s_{nr})\}$.
  supprimer $\langle s_{nl}, s_{nr}, null\rangle$ de $RTStates$
`fin pour` □

On donne en figure 9.6 une représentation graphique de la définition 9.3.2.

La construction d'un graphe qui modélise la suppression est faite au début de la réduction. Dès que `LookForGraphAlternative-e` vérifie qu'il existe des triplets qui représentent des suppressions, ces triplets sont utilisés dans la construction de tous les graphes proposés.

Exemple 9.3.3 Soit $E = a(bc)^+d$ une expression régulière, $w = abcd$ un mot qui appartient à $L(E)$ et $w' = abxyd$ le mot w mis à jour (les symboles x et y ont été insérés et le symbole c a été supprimé). Le graphe de Glushkov sans orbites G_{wo} de E est le suivant :

Avant d'insérer les nœuds qui représentent x et y, `LookForGraphAlternative-e` transforme G_{wo} en un graphe (ci-dessous) dont le nœud qui représente c (nœud 3) est rendu optionnel.



Ainsi, le processus de réduction et de proposition de nouveaux graphes travaille sur le graphe résultant du traitement des triplets de la forme $\langle s_{nl}, s_{nr}, null\rangle$ et les expression régulières candidates seront construites basées sur l'expression $a(bc?)^+d$. □

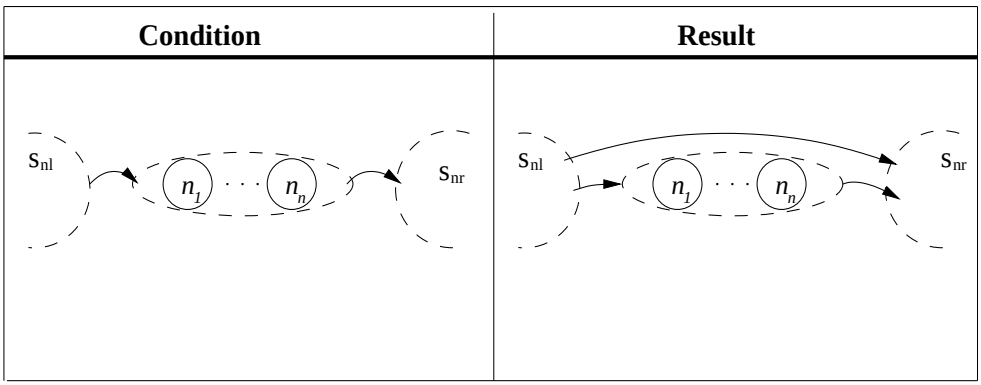


FIG. 9.6 – Conditions et modifications testées par `LookForGraphAlternative-e` pour rendre un ensemble de nœuds optionnels.

Si l'arc inséré depuis s_{nl} vers s_{nr} rend optionnel un groupe de nœuds du graphe à réduire, deux candidates peuvent être générées à partir de ce contexte, l'une dont le groupe est entouré par l'opérateur ? et l'autre dont chaque symbole du groupe est décoré avec ?.

Par exemple, étant donné $E = abcd$, $w = abcd$ et $w' = ad$, les candidats peuvent être $a(bc)?d$ et $ab?c?d$.

Notons que toutes les mises à jours apportées aux graphes doivent être accompagnées par des mises à jour de leurs orbites car les hiérarchies des orbites de chaque graphe doivent respecter

les propriétés des graphes de Glushkov. L'algorithme 8.4.1 utilisé par GREC est utilisé aussi par GREC-e pour maintenir les hiérarchies des orbites conformes aux propriétés de Glushkov.

9.4 Caractéristiques de GREC-e

GREC-e utilise la même approche que GREC pour classer les candidates, *i.e.*, par contexte. Comme les orbites guident toujours le processus de réduction du graphe d'origine, GREC-e les utilise de la même façon pour ce classement. Ainsi, la définition de contexte présentée dans la section 8.5.1 s'applique dans le cas de GREC-e.

Le traitement de l'ambiguïté aussi est le même que celui proposé pour GREC, *i.e.*, lorsque GREC-e détecte, en utilisant la propriété de la proposition 8.5.1, qu'une expression régulière candidate est ambiguë, il le signale à l'utilisateur, lequel peut alors appeler une fonction pour transformer l'expression ambiguë en une expression régulière non ambiguë équivalente, en appliquant par exemple une approche comme celles proposées dans [Aho97, CZP98].

9.4.1 Caractérisation des solutions proposées par GREC-e

Rappelons que, après le traitement des triplets dans $RTStates$, l'algorithme GraphToRegExp est appliqué. Cela peut être fait car toutes les modifications faites sur G_{wo} pour obtenir G_{new} respectent les propriétés d'un graphe de Glushkov. C'est ce que formule le théorème suivant.

Théorème 9.4.1 *Soient G un graphe sans orbite réductible et $\mathcal{H}$ la hiérarchie des orbites de G. Soit R_i l'une des règles de réduction $\boldsymbol{R}_1$, $\boldsymbol{R}_2$ ou $\boldsymbol{R}_3$. Soit $STrans$ l'ensemble de couples (a, e). Pour tous les nœuds s_{nl}, s_{nr} et s_{new} dans les triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ dans $RTStates$, les graphes G_{new} résultants de l'exécution de*
LookForGraphAlternative-e(G, $\mathcal{H}$, R_i, $RTStates$, $STrans$) *sont réductibles.*

Preuve : D'après le théorème 8.5.1, on sait que LookForGraphAlternative retourne une paire $(G_{new}, \mathcal{H}_{new})$ tel que G_{new} est une graphe réductible.
LookForGraphAlternative-e traite plusieurs triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ en respectant la même approche de LookForGraphAlternative. Ainsi, il faut juste démontrer que les définitions de construction de graphes réductibles.
Les définitions utilisées par LookForGraphAlternative-e sont basées aussi sur les définitions proposées pour LookForGraphAlternative : le nœud s_{new} et ses arcs sont insérés de manière similaire. Alors, les graphes proposés par LookForGraphAlternative-e sont aussi toujours réductibles. □

Par ailleurs, soit k est le nombre de triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ dans $RTStates$ tels que $s_{new} \neq null$. On peut établir que GREC-e retourne au moins une expression régulière E' différente de l'expression d'origine E et que E' a au plus k nouveaux symboles par rapport à E, *i.e.*, $\mathcal{D}(E, E') \leq k$. De plus, les langages associés aux solutions données par GREC contiennent au moins le langage d'origine et le nouveau mot. En d'autres termes, GREC-e trouve des solutions (au moins une) correctes par rapport aux objectifs énoncés. C'est ce que formule le théorème suivant.

Théorème 9.4.2 *Soient E une expression régulière et $L(E)$ le langage associé à E. Étant donné $w[0:n] \in L(E)$ $(0 \leq n)$, soit w' un mot résultat d'une suite d'opérations de mises à jour sur des positions p (avec $0 \leq p \leq n$) de w tel que $w' \notin L(E)$. Soient M_E l'automate de Glushkov construit à partir de E et G le graphe obtenu à partir de M_E. Soit $(G_{wo}, \mathcal{H})$ la paire qui représente le graphe sans orbite et la hiérarchie des orbites de G, respectivement. Soient RTStates la relation (non-vide) qui stocke les triplets $\langle s_{nl}, s_{nr}, s_{new} \rangle$ et STrans l'ensemble des arcs (a, e) qui restreint la construction des candidats. Soit k le nombre de triplets $\langle s_{nl}, s_{nr}, s_{new} \rangle$ dans RTStates tel que $s_{new} \neq null$.*
L'exécution de `GREC-e` $(G_{wo}, \mathcal{H}, RTStates, STrans)$ *retourne un ensemble fini et non vide d'expressions régulières candidates $\{E_1, \ldots, E_m\}$. De plus, pour tout $E_i \in \{E_1, \ldots, E_m\}$, nous avons $L(E) \cup \{w'\} \subset L(E_i)$ et $\mathcal{D}(E, E_i) \leq k$.*

Preuve : Voir annexe E □

9.4.2 Discussion sur la complexité et l'implantation

Notre méthode pour guider l'évolution des schémas XML suite à une séquence d'opérations de mises à jour a été basée sur celle qui traite une mise à jour à la fois. Ainsi, `GREC-e` a été proposé en gardant les lignes majeures de `GREC` :

- L'utilisation de l'algorithme de Glushkov pour transformer une expression régulière en automate et donc, des propriétés des automates de Glushkov [CZ00] pour proposer une méthode qui préserve ces propriétés.
- L'utilisation de l'algorithme de réduction qui transforme un automate de Glushkov en une expression régulière [CZ00].

Comme nous avons vu dans la section 8.5.4, `GREC` construit les candidats pour remplacer l'expression régulière d'origine en temps

$$O(x^2 + (x_n^2 \times k)),$$

où x est le nombre de nœuds du graphe d'origine, x_n est le nombre de nœuds d'un des graphes proposé comme candidat et k est le nombre de graphes candidats proposés. Pour `GREC-e`, on ajoute une autre constante qui est le nombre d'appel à `GREC-e` après la construction d'un graphe candidat. Ce nombre d'appels a un rapport avec le nombre de triplets dans la relation *RTStates*. Soit n_t le nombre de triplets de *RTStates*. Alors, la complexité d'exécution de `GREC-e` est en temps

$$O(x^2 + ((x_n^2 \times k) \times n_t)).$$

Remarquer que, si un utilisateur se sert des opérations de mises à jour sur un document XML pour avoir des suggestions de mises à jour des schémas, on pourrait dire que ces opérations auront une sémantique par rapport au schéma d'origine, c'est-à-dire, les insertions et suppressions seront basées sur le besoin de l'utilisateur concernant un nouveau schéma. Ainsi, le nombre de candidats proposés par `GREC-e` peut être inférieur à celui proposé par `GREC` car l'ensemble de mises à jour, en utilisant l'ensemble $STrans$, pourra amener `GREC-e` à ne pas proposer des candidats "inutiles". Cette caractéristique est montrée dans la prochaine section qui présente une exécution pas-à-pas de `GREC-e`.

9.5 Exécution pas à pas

Dans cette section, nous présentons un exemple détaillé de l'exécution de GREC-e. Les mises à jour de cet exemple sont faites toujours sur les fils du nœud étiqueté *Publication* dans la position 02 de l'arbre t dans la figure 7.3. Soit $w = q_{Sujet}\ q_{Annee}\ q_{Revue}$ le mot d'état construit à partir des fils du nœud dans la position 02.
Soit $Publication$, $\langle\emptyset, \emptyset\rangle$, $q_{Sujet}\ (q_{Annee}\ {q_{Revue}}^{+})^{*} \rightarrow q_{Publication}$ la règle de transition à appliquer sur les nœuds étiquetés *Publication*. La figure 9.7 montre le graphe de Glushkov G construit à partir de l'expression régulière E de la règle de transition ci-dessus et son graphe sans orbites G_{wo}, où les nœuds 0, 1, 2, 3 et 4 représentent les états *initial*, q_{Sujet}, q_{Annee}, q_{Revue} et #, respectivement. Remarquons que la hiérarchie des orbites est $\mathcal{O}_1 = \{3\}$, $\mathcal{O}_2 = \{2, 3\}$ et la racine $\mathcal{O}_r = \{0, 1, 2, 3, 4\}$ et que les contextes sont $\mathcal{C}_1 = \{Revue\}$, $\mathcal{C}_2 = \{Annee\}$ et $\mathcal{C}_r = \{Sujet\}$.

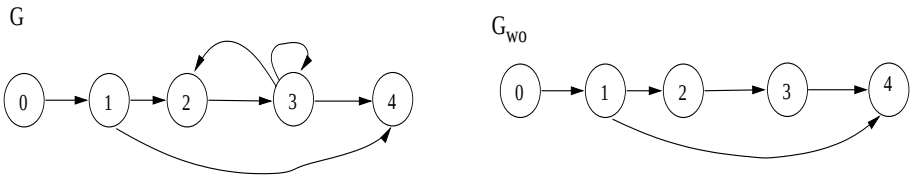


FIG. 9.7 – Le graphe de Glushkov G qui représente $q_{Sujet}\ (q_{Annee}\ {q_{Revue}}^{+})^{*}$ et son graphe sans orbites G_{wo}.

D'abord, considérons que la séquence de mises à jour représentée par la relation *UpdateTable* contient les triplets suivants:

1. $(2, delete, null)$.
2. $(3, insert, q_{Conference})$.
3. $(3, insert, q_{Rapport})$.
4. $(3, insert, q_{Revue})$.
5. $(3, insert, q_{Annee})$.
6. $(3, insert, q_{Rapport})$.
7. $(3, insert, q_{Revue})$.

La table $TMaps$ construite en utilisant l'algorithme 9.2.1 est la suivante :

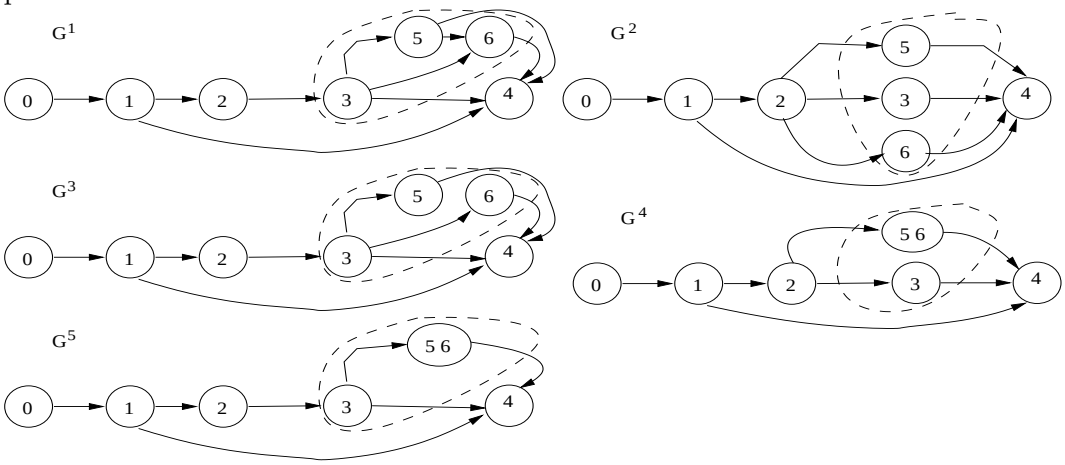
old	*new*
$q_{Sujet}\ q_{Annee}$	$q_{Sujet}\ q_{Annee}$
q_{Revue}	ε
ε	$q_{Conference}\ q_{Rapport}\ q_{Revue}\ q_{Annee}\ q_{Rapport}\ q_{Revue}$
#	#

L'algorithme 9.2.3 construit la relation $RTStates$ et l'ensemble $STrans$ et nous avons :
$RTStates = (\langle 2, 3, null\rangle, \langle 2, 3, 5\rangle, \langle 2, 3, 6\rangle)$ et $STrans = \{(5, q_{Rapport}), (2, q_{Rapport})\}$, où l'état 5 représente le symbole $q_{Conference}$ et l'état 6 le symbole $q_{Rapport}$. Les contraintes imposées par $STrans$ ne permettent pas la construction de candidats qui n'ont pas de transitions $\delta(5, q_{Rapport}) \neq \perp$ et $\delta(2, q_{Rapport}) \neq \perp$ dans leurs automates d'états finis. Remarquer que le triplet qui décrit une suppression $\langle 2, 3, null\rangle$ représente un arc dans G_{wo}, *i.e.*, aucun nœud dans G_{wo} ne va devenir optionnel.

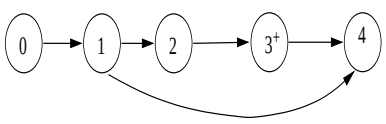
Nous avons vu tous les paramètres d'entrée de GREC-e : G_{wo}, $\mathcal{H}$, $RTStates$ et $STrans$. Pour faciliter la lecture des pas d'exécution nous mettons des indices dans chaque appel à GREC-e, ainsi, $\texttt{GREC-e}_0$ est l'exécution de l'appel principal de GREC-e.

Maintenant, nous considérons des étapes de l'exécution de GREC-e_0 (figure 9.4) :

1. Comme *RTStates* n'est pas vide, la condition dans la ligne 02 n'est pas vérifiée, et G_{wo} a plus d'un nœud (condition de la ligne 05), donc l'exécution GREC-e_0 continue.
2. Le graphe G_{wo} a un nœud qui représente une orbite entière : le nœud 3 (l'orbite $\mathcal{O}_1$) donc l'extension de la définition 8.4.6 est appliquée par *LookForGraphAlternative-e* (ligne 07). Le troisième cas de la définition 8.4.6 est respecté par les deux triplets dans *RTStates* donc *LookForGraphAlternative-e* exécute les pas suivants :
   (a) Elle utilise les triplets $\langle 2,3,5\rangle$ et $\langle 2,3,6\rangle$ pour construire les graphes qui sont proposés par le troisième cas de la définition étendue :



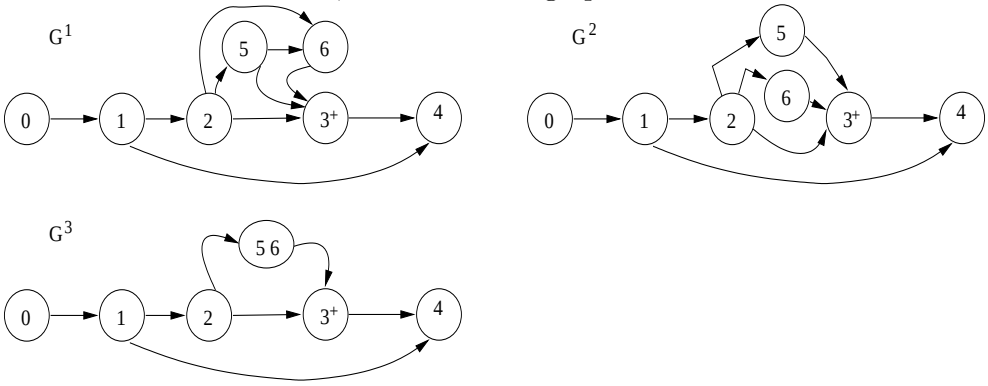
   Des graphes présentés ci-dessus, seuls les graphes G^2 et G^4 sont proposés car ils respectent les contraintes imposées dans l'ensemble $STrans$, *i.e.*, ils ont les arcs $(5,6)$[5] et $(2,5)$ représentés par les paires $(5, q_{Rapport})$ et $(2, q_{Rapport})$, respectivement. Les triplets $\langle 2,3,5\rangle$ et $\langle 2,3,6\rangle$ sont supprimés de *RTStates* donnant *RTStates'*.
3. Le résultat de l'exécution de *LookForGraphAlternative-e* dans GREC-e_0 est constitué de 2 nouveaux graphes, alors GREC-e est rappelé pour chaque graphe.
4. GREC-e_1 prend les paramètres G^2, $\mathcal{H}'$ (avec les nouvelles orbites), *RTStates'* et *STrans*:
   (a) Comme *RTStates'* ne contient plus de triplets, la ligne 03 est exécutée par GREC-e_1.
   (b) La procédure *GraphToRegExp* est appelée et le graphe en entrée est transformé en l'expression régulière $q_{Sujet}\ (q_{Annee}\ (q_{Revue}|q_{Conference}|q_{Rapport})^+)^*$ puis l'exécution de GREC-e_1 termine.
5. GREC-e_2 prend les paramètres G^4, $\mathcal{H}'$ (avec les nouvelles orbites), *RTStates'* et *STrans*:
   (a) En utilisant les mêmes pas que ci-dessus, la procédure *GraphToRegExp* est appelée et le graphe en entrée est transformé en l'expression régulière $q_{Sujet}\ (q_{Annee}\ (q_{Revue}\ |(q_{Conference}!\ q_{Rapport}!))^+)^*$, puis GREC-$e_2$ termine.
6. Les expressions candidates sont stockées dans le contexte $\mathcal{C}_1$, car l'orbite utilisée est l'orbite $\mathcal{O}_1$.
7. L'exécution de GREC-e_0 continue : la fonction *ApplyRule* applique la décoration sur le graphe G_{wo} et nous avons G'_{wo}:



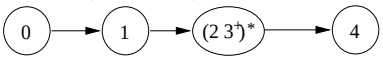
   et $\mathcal{H}_1 = \{\{2,3\},\{0,1,2,3,4\}\}$.

[5]Remarquer que le graphe G^2 n'a pas l'arc $(5,6)$ car les arcs qui représentent les orbites n'apparaissent pas dans G^2 mais, en utilisant la hiérarchie des orbites de G^2, on vérifie que cet arc existe.

8. GREC-e est rappelé et nous avons GREC-e$_3$ qui reçoit comme paramètres le nouveau graphe G'_{wo}, la nouvelle hiérarchie des orbites $\mathcal{H}'$ et la relation $RTStates$ et l'ensemble $STrans$ originaux. L'exécution de GREC-e$_0$ finit lorsque l'exécution de GREC-e$_3$ finit aussi.
   (a) La condition de la ligne 02 n'est pas satisfaite donc, la fonction *ChooseRule* retourne la règle $\mathbf{R}_1$ à exécuter sur les nœuds 2 et 3.
   (b) *LookForGraphAlternative-e* vérifie si l'un des cas de la définition 9.3.1 est respecté. C'est le cas du *Cas 1* pour les deux triplets $\langle 2,3,5\rangle$ et $\langle 2,3,6\rangle$. En suivant le premier cas de la définition 9.3.1, nous avons les graphes :



   Les graphes proposés (*i.e.*, ceux qui respectent les paires dans $STrans$) sont G^1 et G^2. Alors, GREC-e est appelé 2 fois avec les nouveaux paramètres (remarquons que la relation $RTStates'$ ne contient plus de triplets). Donc, les 2 appels construisent 2 nouveaux candidats stockés dans le contexte $\mathcal{C}_2$ qui représente l'orbite $\mathcal{O}_2$.
      i. GREC-e$_4$ (avec G^1) retourne $q_{Sujet}\ (q_{Annee}\ q_{Conference}!\ q_{Rapport}!\ {q_{Revue}}^+)^*$
      ii. GREC-e$_5$ (avec G^2) retourne[6] $q_{Sujet}\ (q_{Annee}\ (q_{Rapport}|q_{Conference})^*\ {q_{Revue}}^+)^*$
   (c) L'exécution de GREC-e$_3$ passe à la ligne 09 et la règle $\mathbf{R}_1$ est appliquée sur le graphe G_{wo} donnant un graphe G'_{wo} et une nouvelle hiérarchie d'orbites $\mathcal{H}'$.
9. GREC-e est rappelé et nous avons GREC-e$_6$ avec les nouveaux paramètres.
   (a) Les conditions des lignes 02 et 05 ne sont pas vérifiées, alors GREC-e$_6$ continue.
   (b) Le nœud qui représente $\mathcal{O}_2$ sera décoré avec $^+$ car il représente l'orbite entière.
   (c) *LookForGraphAlternative-e* vérifie si l'une des conditions dans la définition 8.4.6 étendue est respectée. Aucune condition n'est respectée.
10. GREC-e$_6$ continue l'exécution et *ApplyRule* décore le nœud 2 3^+ avec un + et nous avons donc $(2\ 3^+)^+$
11. GREC-e$_6$ rappelle GREC-e avec le nouveau graphe et la hiérarchie des orbites $\mathcal{H} = \{0,1,2,4\}$ (remarquons que le nœud 3 a été fusionné avec le nœud 2): c'est GREC-e$_7$.
12. GREC-e$_7$ vérifie si l'ensemble $STStates$ est vide. La condition n'est pas respectée alors l'exécution continue. G_{wo} a encore 4 nœuds alors la condition de la ligne 05 n'est pas respectée non plus.
13. *ChooseRule* choisit la règle $\mathbf{R}_2$ à appliquer sur le nœud 2.
14. *LookForGraphAlternative-e* ne propose aucun graphe car les conditions de la définition 8.4.5 étendue ne sont pas respectées.
15. *ApplyRule* modifie le graphe en entrée (ligne 09) et nous avons G_{wo} :



[6] La solution avec l'opérateur ? ne peut pas être proposée car il faut qu'il existe un arc depuis $q_{Conference}$ vers $q_{Rapport}$.

16. Une nouvelle exécution de GREC-e est initialisée (GREC-e$_8$)
17. La relation *RTStates* n'est pas vide alors la condition dans la ligne 02 n'est pas respectée ni la condition de la ligne 05, alors GREC-e$_8$ continue.
18. *ChooseRule* choisit la règle $\mathbf{R}_1$ à appliquer sur les nœuds 2 et 4.
19. *LookForGraphAlternative-e* vérifie si l'une des conditions de la définition 9.3.1 est respectée. Aucune condition n'est respectée.
20. La règle $\mathbf{R}_1$ est appliquée sur le graphe G_{wo} et la hiérarchie des orbites $\mathcal{H}$ est mise à jour aussi.
21. Remarquons qu'à ce moment-ci, aucune nouvelle exécution ne va plus proposer de candidats car les nœuds s_{nl} et s_{nr} dans les triplets de *RTStates* ont déjà été fusionnés par le processus de réduction.
22. Alors l'exécution GREC-e$_8$ finit, puis l'exécution GREC-e$_7$, puis l'exécution GREC-e$_6$ et ainsi de suite, jusqu'à terminer l'exécution GREC-e$_0$.

Les candidats proposés classés par contexte sont:

Revue	1.	$q_{Sujet}\ (q_{Annee}\ (q_{Revue}\|q_{Conference}\|q_{Rapport})^+)^*$
	2.	$q_{Sujet}\ (q_{Annee}\ (q_{Revue}\ \|(q_{Conference}!\ q_{Rapport}!))^+)^*$
Annee	1.	$q_{Sujet}\ (q_{Annee}\ q_{Conference}!\ q_{Rapport}!\ {q_{Revue}}^+)^*$
	2.	$q_{Sujet}\ (q_{Annee}\ (q_{Rapport}\|q_{Conference})^*\ {q_{Revue}}^+)^*$
Sujet		

Si l'on considère que les nouveaux symboles doivent être insérés dans le même contexte que *Revue*, ce qui est raisonnable du fait que les articles de conférences, de revues et rapport de recherche doivent être modélisés ensemble, alors, l'utilisateur fait son choix parmi les options du contexte *Revue*. Si son choix est la première option de ce contexte, alors elle peut remplacer le modèle de contenu de l'élément :

```
<!ELEMENT Publications (Sujet,(Annee,Revue+)*)>
```

par le nouveau modèle :

```
<!ELEMENT Publication (Sujet,(Annee,(Revue|Conference|Rapport)+)*)>
```

9.6 Discussion

Dans ce chapitre nous avons présenté l'approche qui étend GREC pour traiter des mots qui ont été mis à jour dans plusieurs positions. En prenant les définitions utilisées par GREC, nous proposons GREC-e. Bien que GREC-e est une algorithme plus coûteux que GREC, il se sert de l'ensemble de mises à jour pour restreindre certains candidates et, comme montré dans section 9.5, le nombre d'expressions régulières candidates engendrées par GREC-e peut être moins important que le nombre engendré par GREC (section 8.6).

Pour réussir à transformer la version GREC qui traite un mot avec une seule position mise à jour en GREC-e qui traite un mot avec plusieurs positions mises à jour, nous avons créé une structures ($TMaps$) qui permet de trouver à partir de la relation des opérations de mises à jour, les endroits exacts d'où chercher les symboles du nouvel mot qui rendent ce mot invalide. Cette structure permet aussi de construire la relation qui stocke les triplets qui représentent le nouveau symbole à insérer et la position d'insertion dans le graphe d'origine qui seront passés à GREC-e pour construire les candidats.

L'avantage de se baser sur GREC pour traiter les mises à jour multiples est que nous pouvons utiliser ce qui était déjà défini :

- La classification des candidats (contextes).
- La vérification d'ambiguïté des candidats.
- Les définitions de constructions des candidats, il a fallu définir seulement les extensions des définitions pour GREC.

Bien que l'inférence grammaticale ne s'adapte pas à notre problème, elle s'adapte bien à des autres problèmes dans le cadre de XML. Par exemple, Dans [Chi01, Chi02, GGR$^+$00, Fer01], il est proposé des méthodes d'extraction des schémas à partir de documents XML en utilisant l'approche de l'inférence grammaticale.

Quatrième partie

Conclusions et annexes

Chapitre 10

Conclusions et perspectives

Une méthode d'évolution de schémas XML qui préserve la cohérence des documents sans les modifier est très utile à des applications qui utilisent le langage XML pour modéliser les données. D'une part les données (documents) préexistantes restent valides par rapport au nouveau schéma, d'autre part aucun processus de revalidation de toute la base de données n'est nécessaire ce qui épargne du temps et évite des erreurs. Ainsi, dans cette thèse, nous avons présenté une approche pour l'évolution de schémas XML préservant la validité des documents. Pour cela nous avons proposé et/ou utilisé des méthodes :
(*i*) de validation des documents XML qui s'appuie sur les automates d'arbres (chapitre 7),
(*ii*) de validation incrémentale suite à une mise à jour du document XML (chapitre 4), et
(*iii*) de réduction d'un automate en l'expression régulière correspondante.
L'automate d'arbre utilisé dans la validation de documents XML (vus comme des arbres) se sert des expression régulières pour associer un état à un nœud de l'arbre et ainsi, les automates d'états finis (les automates de Glushkov) ont été la base de notre approche.

Pour construire notre méthode d'aide à l'évolution des schémas, nous avons donc utilisé et/ou proposé des algorithmes pour :

- Transformer un schéma en un automate d'arbre.
- Transformer un graphe de Glushkov en un graphe de Glushkov sans orbites et une hiérarchie des orbites.
- Réduire un graphe de Glushkov sans orbites en l'expression régulière correspondante. Cet algorithme a été basé sur les définitions de réduction dans [CZ00].

Ces algorithmes ont été la base pour notre l'algorithme d'aide à l'évolution des schémas `GREC` (et `GREC-e`). La complexité de `GREC`, en temps d'exécution, est en $O(x^2 + (x_n^2 \times m))$, où x est le nombre de nœuds du graphe d'origine, x_n est le nombre moyen de nœuds des graphes candidats à réduire et m est le nombre de graphes à réduire. L'algorithme `GREC` a été implanté en ASF+SDF et un prototype simplifié (qui ne traite que les règles $\mathbf{R}_1$ et $\mathbf{R}_2$) de l'algorithme `GREC-e` a été implanté en Java.

La motivation de proposer `GREC` (et `GREC-e`) est fondée sur le fait que les administrateurs des applications XML peuvent avoir besoin d'un outil pour les aider dans le processus d'évolution des schémas puisque cette tâche n'est pas triviale à mesure qu'il faut prendre en compte tous les documents de la base de données. Si l'utilisateur n'est pas un expert en informatique, alors il peut ajouter des nouveaux éléments dans le schéma de façon que le résultat de la mise à jour ne préserve pas la cohérence des documents préexistants. Dans ce cas, une méthode qui s'appuie sur un document exemple pour étendre une classe de documents préexistante est très utile car l'utilisateur aura des modèles de schéma qui préservent la validité des documents pour remplacer le schéma d'origine. Ce document exemple peut être un document auparavant valide qui, après

une mise à jour, ne respecte plus son schéma.

La méthode de validation des documents XML introduite dans le chapitre 7 et les principes de validation incrémentale présentés dans [BH03] ont été la base de notre approche pour guider l'évolution incrémentale de schémas. Si des mises à jours exécutées par un administrateur sur un arbre XML t rendent impossible le succès de l'exécution de l'automate d'arbre $\mathcal{A}$, le processus de guidage de l'évolution est déclenché.

La méthode d'évolution prend alors comme paramètres d'entrée les informations de la règle de transition de $\mathcal{A}$ qui n'a pas pu associer un état à un nœud de l'arbre et les informations de mise à jour. Elle peut être résumée comme suit :

- L'utilisateur exécute une séquence de mises à jour sur un document XML X valide en envisageant de faire évoluer le schéma selon les caractéristiques du document mis à jour.
- L'une des règles de transition $a, S, E \rightarrow q_a$ de l'automate d'arbre échoue lors de l'association d'un état à un nœud concerné par les mises à jour.
- Si la vérification des attributs échoue, alors le couple $S = \langle S_{compulsory}, S_{optional} \rangle$ est mis à jour : soit les nouveaux attributs sont insérés dans l'ensemble $S_{optional}$, soit les attributs manquants sont déplacés vers l'ensemble $S_{optional}$.
- Si l'automate d'états finis de E n'accepte pas le mot construit à partir de la concaténation des fils du nœud concerné, alors de nouvelles expressions régulières sont proposées pour remplacer E.
- Le schéma est mis à jour de façon incrémentale et tous les documents préexistants dans la base de données restent valides par rapport au nouveau schéma construit.

Dans cette thèse, nous avons proposé deux méthodes pour guider l'évolution des schémas qui préserve la validité des documents préexistants sans les modifier : l'une qui est fondée sur une seule mise à jour (`GREC`) et l'autre fondée sur des mises à jour multiples (`GREC-e`). L'approche proposée utilise les DTD comme langages de schémas car elle est basée sur la méthode de validation. En revanche, en ce qui concerne `GREC` (ou `GREC-e`) le langage de schéma n'a pas d'impact (du moment qu'il s'appuie sur des expressions régulières).

À notre connaissance, il n'existe pas de méthode de guidage de l'évolution des schémas XML similaire à la méthode proposée dans cette thèse. La majorité des méthodes qui traitent les documents associés à schémas, comme le montrent le chapitre 5, propose l'adaptation de documents valides par rapport à un schéma d'origine à un schéma cible. De plus, ces méthodes n'ont pas été conçues pour l'évolution des schémas. Nous avons montré aussi des méthodes qui proposent des primitives de mises à jour des schémas. Ces méthodes sont différentes de notre proposition car l'utilisateur exécute la mise à jour directement dans le schéma et il peut aussi survenir des pertes de données des documents XML pendant le processus de mise à jour du schéma. Par ailleurs, il faut que l'utilisateur ait une connaissance syntaxique du schéma pour proposer la mise à jour.

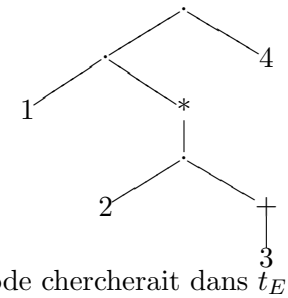
Dans notre méthode pour guider l'évolution des schémas nous considérons plusieurs directions de recherche à travailler :

– Généraliser la méthode pour qu'elle prenne en considération différents mots appartenant à différents nœuds père de l'arbre, mais ces pères ont la même étiquette. Plus précisément, leurs fils sont contraints par la même expression régulière (règle de transition). Dans ce contexte, nous aurons plusieurs paires (*mot d'origine* w, *mot d'origine mise à jour* w'). Pour cela nous envisageons de modifier les processus de préparation de données (section 9.2) pour traiter un ensemble de paires (w,w'). Ensuite `GREC-e` s'appliquera sans changement.

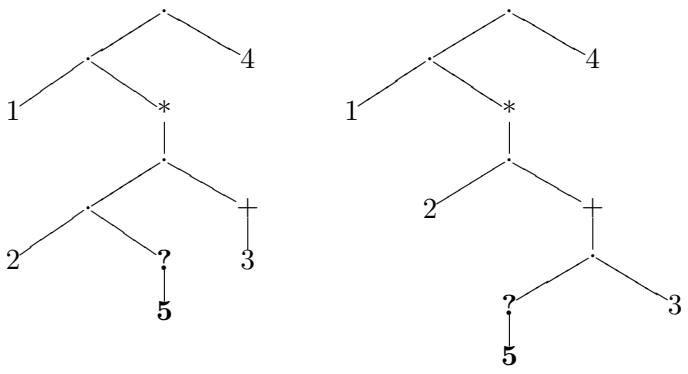
– Étudier aussi une autre mesure de distance qui prend en considération tous les symboles des candidats, *i.e.*, parenthèses, opérateurs, les nouvelles positions des symboles d'origine, entre autres. Le but serait de mesurer l'impact de la modification de l'expression régulière d'origine pour trouver une expression régulière candidate. En s'appuyant sur cette mesure, nous pourrons étudier des mécanismes pour classer les schémas candidats dans le but de ne pas proposer des schémas trop généralisés. Par exemple, si une opération d'insertion insère des éléments d'une façon que une expression régulière $E = (abc)^+$ devienne $E' = (a|b|c)^*$, alors E' peut ne pas être proposée. Ce travail suppose des techniques de comparaison - classification d'expressions régulières : il pourrait avoir pour application, au-delà de `GREC`, l'integration de différents schémas.

– Étendre un langage de mise à jour existant (*e.g.*, UpdateX [SHS04]) pour traiter notre méthode d'évolution de schéma. Le but est de construire une interface pour l'utilisation de notre méthode.

– Considérer l'expression régulière comme un arbre syntaxique et utiliser cet arbre pour construire les candidats. Dans ce contexte, les nœuds de l'arbre d'abstraction contiennent les symboles de l'expression régulière et les opérateurs. Par exemple, l'arbre t_E ci-dessous représente l'expression indicée $\overline{E} = 1\ (2\ 3^+)^*4$:



Dans cette approche, la méthode chercherait dans t_E les positions où le nouveau symbole pourrait être inséré.
Dans [ZPC97], les auteurs utilisent un arbre syntaxique associé à l'expression E pour appliquer les fonctions de Glushkov et, ensuite, construire les automates de Glushkov. Ainsi, en s'appuyant sur leur travail on pourrait construire des arbres candidats à partir de t_E. Nous pensons que le coût de construction des candidates pourra être moins importante que travailler sur le graphe de Glushkov car on n'utilise pas un processus de réduction et, par conséquent, on ne construit ni la hiérarchie des orbites ni le graphe sans orbites. En effet, on construirait les nouveaux arbres syntaxiques à partir de l'arbre d'origine selon les cas à appliquer. Par exemple, reprenons l'arbre t_E ci-dessus, supposons que $s_{nl} = 2$ et $s_{nr} = 3$, le nœud s_{new} (*e.g.*, 5) le nœud à insérer. Les arbres ci-dessous montrent les nouveaux arbres syntaxiques :



Les expressions régulières sont $\overline{E} = 1\ (2\ 5?\ 3^+)^*4$ et $\overline{E} = 1\ (2\ (5?\ 3)^+)^*4$, respectivement.

– Étudier une caractérisation de l'apprentissage pour les automates de Glushkov et ainsi, appliquer l'approche d'inférence grammaticale pour trouver les candidats ou le candidat idéal. Nous envisageons, dans ce cas, de changer l'algorithme RPNI pour qu'il préserve les propriétés de Glushkov pendant les pas de construction de l'automate cible : la fusion des états et la déterminisation de l'automate.

– Utiliser notre méthode dans le même cadre que celui de la méthode proposée dans [BGMT02] (section 5.3) : adapter le schémas selon un nouveau document qui arrive dans la base de données. Pour cela, le document X à insérer dans la base de données XML est comparé avec les documents préexistants et, en utilisant le document X' le plus proche de X, on construit une séquence de mises à jour sur X' qui résulte X. Ensuite, l'approche `GREC-e` pourrait être appliquée afin de proposer des schémas pour remplacer le schéma d'origine.

– Appliquer la méthode dans l'ensemble du schéma (non plus une seule expression régulière) : dans ce cas les mots à traiter seront représentés par des arbres. Dans ce contexte, l'impact du langage de schéma peut être très important. Pour traiter ce problème il faut envisager des solutions différentes de celles utilisées dans `GREC`.

Bibliographie

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web - From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.

[AE03] L. Al-Jadir and F. El-Moukaddem. Once upon a time a dtd evolved into another dtd... In *International Conference Object-Oriented Information Systems*, volume 2817 of *Lecture Notes in Computer Science*. Springer, 2003.

[Age04] Clever Age. Faisons le point sur les langages de schéma XML. In *http://www.clever-age.com/*, 2004.

[Aho97] H. Ahonen. Disambiguation of SGML content models. In *PODP*, volume 1293 of *Lecture Notes in Computer Science*. Springer, 1997.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

[AMN+01] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *ACM Symposium on Principles of Database System*, 2001.

[Ang82] D. Angluin. Inference of reversible languages. *J. ACM*, 29(3):741–765, 1982.

[BB00] J. Berstel and L. Boasson. XML grammars. In *Mathematical Foundations of Computer Science*, number 1893 in LNCS, pages 182 – 191, 2000.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. *SIGPLAN Not.*, 38(9):51–63, 2003.

[BCT04] N. Bidoit, S. Cerrito, and V. Thion. A first step towards modelling semistructured data in hybrid multi-modal logic. *Journal of Applied Non-Classical Logics*, 2004. To appear.

[BDH+04a] B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, D. Laurent, and M. A. Musicante. Learning from failure: Conservative extension of regular languages. In *International Conference of the Chilean Computer Science Society*, number PR2200 in IEEE Proceedings, pages 99–109, 2004.

[BDH+04b] B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, D. Laurent, and M. A. Musicante. Schema evolution for XML: A consistency-preserving approach. In *Mathematical Foundations of Computer Science*, number 3153 in LNCS, pages 876 – 888, 2004.

[BDH04c] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287, 2004.

[BDHL03] B. Bouchou, D. Duarte, M. Halfeld Ferrari Alves, and D. Laurent. Extending tree automata to model XML validation under element and attribute constraints. In *ICEIS*, 2003.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. *SIGMOD Rec.*, 25(2):505–516, 1996.

[BGMT02] E. Bertino, G. Guerrini, M. Mesiti, and L. Tosetto. Evolving a set of DTDs according to a dynamic set of XML documents. In *XML-Based Data Management - EDBT*, volume 2490 of *Lecture Notes in Computer Science*. Springer, 2002.

[BH03] B. Bouchou and M. Halfeld Ferrari Alves. Updates and incremental validation of XML documents. In *The 9th DBPL*, number 2921 in LNCS, 2003.

[BHKO02] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling rewrite systems: The ASF+SDF compiler. *ACM, Transactions on Programming Languages and Systems*, 24, 2002.

[BK93] A. Bruggeman-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120:197–213, 1993.

[BM04] P. Biron and A. Malhorta. *XML Schema Part 2: Datatypes Second Edition, Recomendation W3C*. http://www.w3.org/TR/XMLschema-2/, 2004.

[BML+04] D. Barbosa, A.O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient incremental validation of XML documents. In *ICDE*, pages 671–682, 2004.

[BNB04] G. J. Bex, F. Neven, and J. V. Bussche. DTDs versus XML schema: a practical study. In *Proceedings of the 7th International Workshop on the Web and Databases*, pages 79–84. ACM Press, 2004.

[BPSM98] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. *Extensible markup language (XML^{TM} 1.0)*. http://www.w3.org/TR/1998/REC-xml-19980210.html, 1998.

[BPV04] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems*, 29(4):710–751, 2004.

[BS86] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theorical Computer Science*, 48(1):117–126, 1986.

[BVPK04] A. Boukottaya, C. Vanoirbeek, F. Paganelli, and O. A. Khaled. Automating XML document transformations: A conceptual modelling based approach. In Sven Hartmann and John F. Roddick, editors, *First Asia-Pacific Conference on Conceptual Modelling (APCCM2004)*, volume 31 of *CRPIT*, pages 81–90, Dunedin, New Zealand, 2004. ACS.

[BW92] A. Bruggeman-Klein and D. Wood. Deterministic regular languages. In *STACS*, 1992.

[BW98a] A. Bruggeman-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998.

[BW98b] A. Bruggeman-Klein and D. Wood. Regular tree languages over non-ranked alphabets. Technical report, unpublished manuscript, 1998.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Available on: *http://www.grappa.univ-lille3.fr/tata*, 1997.

[CDL99] D. Calvanese, G. De Giacomo, and M. Lenzerini. Representing and reasoning on XML documents: A description logic approach. *Journal of Logic and Computation*, 9(3):295–318, 1999.

[CF03] P. Caron and M. Flouret. From Glushkov WFAs to rational expressions. In Zoltán Ésik and Zoltán Fülöp, editors, *Developments in Language Theory*, volume 2710 of *Lecture Notes in Computer Science*. Springer, 2003.

[CFR+01] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery : A query language for XML. Technical report, Available from http://www.w3.org/TR/xquery.html, 2001.

[Chi00] B. Chidlovskii. Using regular tree automata as XML schemas. In *Proc. IEEE Advances in Digital Libraries Conference*, May 2000.

[Chi01] B. Chidlovskii. Schema extraction from XML: A grammatical inference approach. In *International Workshop on Knowledge Representation meets Databases*, volume 45 of *CEUR Workshop Proceedings*. Technical University of Aachen (RWTH), 2001.

[Chi02] Boris Chidlovskii. Schema extraction from xml collections. In *JCDL '02: Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pages 291–292. ACM Press, 2002.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[Cho02] B. Choi. What are real dtds like? In *WebDB*, pages 43–48, 2002.

[Cla99] J. Clark. XSL transformation (XSLT) specification. In *http://www.thaiopensource.com/trex*, 1999.

[CLR00] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1900.

[CM01] J. Clark and M. Murata. *RELAX NG Specification*. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html, 2001.

[CM03] J. Coelho and M.Florido. Type-based XML processing in logic programming. In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 273–285. Springer-Verlag, 2003.

[CNT04] J. Carme, J. Niehren, and M. Tommasi. Querying unranked trees with stepwise tree automata. In *Rewriting Techniques and Applications: 15th International Conference*, volume 3091 of *Lecture Notes in Computer Science*. Springer, 2004.

[Coo03] S. V. Coox. Axiomatization of the evolution of XML database schema. *Programming and Computing Software*, 29(3):140–146, 2003.

[CP92] C.H. Chang and R. Paige. From regular expression to DFA's using NFA's. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, number 664 in Lecture Notes in Computer Science, pages 90–110, Tucson, AZ, 1992. Springer-Verlag, Berlin.

[CR04] C. Chitic and D. Rosu. On validation of XML streams using finite state machines. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 85–90. ACM Press, 2004.

[CS00] R. Costello and J. C. Schneider. Challenge of XML schemas - schema evolution. In *Proceedings of XML Europe*, 2000.

[CZ00] P. Caron and D. Ziadi. Characterization of Glushkov automata. *TCS: Theorical Computer Science*, 233:75–90, 2000.

[CZP98] J.-M. Champarnaud, D. Ziadi, and J.-L. Ponty. Determinization of glushkov automata. In *Third International Workshop on Implementing Automata - WIA'98*, volume 1660 / 1999 of *LNCS*, 1998.

[DL03] S. Dal Zilio and D. Lugiez. XML schema, tree logic and sheaves automata. In *RTA 2003 – 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 246–263. Springer-Verlag, June 2003.

[DM98] P. Dupont and L. Miclet. Inférence grammaticale régulière : fondements théoriques et principaux algorithmes. Technical Report 3449, Institut National De Recherche en Informatique et en Automatique - INRIA - Rennes, 1998.

[DMV94] P. Dupont, L. Miclet, and E. Vidal. What is the search space of the regular inference? In *ICGI '94: Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 25–37. Springer-Verlag, 1994.

[Dup96] P. Dupont. Incremental regular inference. In *International Colloquium on Grammatical Inference: Learning Syntax from Sentences*, Lecture Notes in Computer Science, pages 222–237. Springer-Verlag, 1996.

[Fer01] H. Fernau. Learning XML grammars. In *MLDM '01: Proceedings of the Second International Workshop on Machine Learning and Data Mining in Pattern Recognition*, LNCS, pages 73–87. Springer-Verlag, 2001.

[FGK03] M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees (extended abstract). In *18th Annual IEEE Symposium on Logic in Computer Science - LICS*, pages 188–197, 2003.

[FL01] Wenfei Fan and Leonid Libkin. On XML integrity constraints in the presence of DTDs. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 114–125. ACM Press, 2001.

[FSW99] M. Fernandez, Jérôme Siméon, and Philip Wadler. XML query languages: Experiences and exemplars. Technical report, Available from http://www.w3.org/1999/09/ql/docs/xquery.html, 1999.

[GGR+00] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: a system for extracting document type descriptors from XML documents. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 165–176. ACM Press, 2000.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[Gol91] C. F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1991.

[GPWZ01] D. Giammarresi, J.-L. Ponty, D. Wood, and D. Ziadi. Thompson digraphs: A characterization. In *Automata Implementation : 4th International Workshop on Implementing Automata*, volume 2214 of *Lecture Notes in Computer Science*, pages 91–100. Springer-Verlag, 2001.

[GPWZ04] D. Giammarresi, J.-L. Ponty, D. Wood, and D. Ziadi. A characterization of thompson digraphs. *Discrete Appl. Math.*, 134(1-3):317–337, 2004.

[GS97] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3. Springer Verlag, 1997.

[HM02] H. Hosoya and M. Murata. Validation and boolean operations of attribute-element constraints. In *Programming Languages Technologies for XML, PLAN-X*, 2002.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 2nd edition, 2001.

[HP03] H. Hosoya and B. C. Pierce. XDuce: A statically typed xml processing language. *ACM Trans. Inter. Tech.*, 3(2):117–148, 2003.

[IDD04] I. E. Iacob, A. Dekhtyar, and M. I. Dekhtyar. Checking potential validity of XML documents. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 91–96. ACM Press, 2004.

[Kil99] P. Kilpelainen. SGML and XML content models. Technical report, Department of Computer Science - University of Helsinki, 1999.

[Kle65] S. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42, 1965.

[KLP02] E. Kuikka, P. Leinonen, and M. Penttonen. Towards automating of document structure transformations. In *ACM - Symposium on Document Engineering, McLean, Virginia, USA, p. 103-110*, 2002.

[KSR02] B. Kane, H. Su, and E. A. Rundensteiner. Consistently updating XML documents using incremental constraint check queries. In *Fourth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2002)*. ACM, 2002.

[KSS03] N. Klarlund, T. Schwentick, and D. Suciu. XML: Model, schemas, types, logics, and queries. In Jan Chomicki, Ron van der Meyden, and Gunter Saake, editors, *Logics for Emerging Applications of Databases*. Springer, 2003.

[LC00] D. L. and W. W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3):76–87, 2000.

[Lei03] P. Leinonen. Automating XML document structure transformations. In *DocEng '03: Proceedings of the 2003 ACM symposium on Document engineering*, pages 26–28. ACM Press, 2003.

[LMM00] D. Lee, M. Mani, and M. Murata. Reasoning about XML schemas languages using formal language theory. In *Technical Report - IBM Almaden Research Center. Available in www.cobase.cs.ucla.edu/tech-docs/˜ dongwon/ibm-tr-2000.ps*, 2000.

[MBPS05] S. Maneth, A. Berlea, T. Prest, and H. Seidl. XML type checking with macro tree transducer. In *PODS*, 2005.

[Meg00] D. Megginson. *SAX 2.0: the Simple API for XML*. Official Website for SAX, http://www.saxproject.org/, 2000.

[MH03] M. Murata and H. Hosoya. Validation algorithm for attribute-element constraints of RELAX NG. In *Extreme Markup Languages*, 2003.

[MLM01] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Language, Montreal, Canada*, 2001.

[MLMK04] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema language using formal language theory. *ACM Transactions on Internet Technology (TOIT)*, To Appear, 2004.

[MN03] W. Martens and F. Neven. Typechecking top-down uniform unranked tree transducers. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory - ICDT 2003*, volume 2572 of *LNCS*. Springer, 2003.

[MSV00] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *ACM Symposium on Principles of Database System*, pages 11–22, 2000.

[Mur95] M. Murata. Forest-regular languages and tree-regular languages. In *Unpublished manuscript*, 1995.

[Mur98] M. Murata. Data model for document transformation and assembly. In *PODDP '98: Proceedings of the 4th International Workshop on Principles of Digital Document Processing*, pages 140–152. Springer-Verlag, 1998.

[Nev99] F. Neven. Extensions of attribute grammars for structured document queries. In *DBPL '99: Revised Papers from the 7th International Workshop on Database Programming Languages*, pages 99–116. Springer-Verlag, 1999.

[Nev02a] F. Neven. Automata, logic and XML. In Julian C. Bradfield, editor, *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*, volume 2471 of *Lecture Notes in Computer Science*. Springer, 2002.

[Nev02b] F. Neven. Automata theory for XML researchers. *Sigmod Record*, 31(3), 2002.

[OG92] J. Oncina and R. Garcia. Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, pages 49 – 61, 1992.

[PGW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260. IEEE Computer Society, 1995.

[PH00] R. Parekh and V. Honavar. Automata induction, grammar inference, and language acquisition. In Robert Dale, H. L. Somers, and Hermann Moisl, editors, *Handbook of Natural Language Processing*. Marcel Dekker, Inc., 2000.

[PV00] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *ACM Symposium on Principles of Database System*, pages 35–46, 2000.

[PV03] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *International Conference on Database Theory - ICDT*, volume 2572 of *Lecture Notes in Computer Science*. Springer, 2003.

[RAJB+00] J. Roddick, L. Al-Jadir, L. Bertossi, M. Dumas, F. Estrella, H. Gregersen, K. Hornsby, J. Lufter, F. Mandreoli, T. Männistö, E. Mayol, and L. Wedemeijer. Evolution and change in data management - issues and directions. *SIGMOD Record*, 29(1):21–25, 2000.

[RB01] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal.*, 10(4):334–350, 2001.

[Roi99] C. Roisin. *Documents multimédia structurés*. habilitation à diriger des recherches, INP Grenoble, 1999.

[SHS04] G. M. Sur, J. Hammer, and J. Siméon. An XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.

[SKC+01] H. Su, D. Kramer, L. Chen, K. T. Claypool, and E. A. Rundensteiner. XEM: Managing the evolution of XML documents. In *Eleventh International Workshop on Research Issues in Data Engineering on Document Management for Data Intensive Business and Scientific Applications*, pages 103–110. IEEE Computer Society, 2001.

[SKR01] H. Su, H. Kuno, and E. A. Rundensteiner. Automating the transformation of XML documents. In *3rd WIDM 2001*. ACM, 2001.

[Suc98] D. Suciu. Semistructured data and XML. In *Proceedings of International Conference on Foundations of Data Organization*, 1998.

[Suc01] D. Suciu. On database theory and XML. *SIGMOD Record*, 30(3), 2001.

[Suc02a] D. Suciu. Typechecking for semistructured data. In Giorgio Ghelli and Gösta Grahne, editors, *DBPL*, volume 2397 of *LNCS*. Springer, 2002.

[Suc02b] D. Suciu. The XML typechecking problem. *SIGMOD Rec.*, 31(1):89–96, 2002.

[SV02] L. Segoufin and V. Vianu. Validating streaming XML documents. In *ACM Symposium on Principles of Database System*, 2002.

[TBMM04] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures Second Edition, Recomendation W3C*. http://www.w3.org/TR/XMLschema-1/, 2004.

[Tho97] W. Thomas. Languages, automata and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3. Springer Verlag, 1997.

[Tic84] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.*, 2(4):309–321, 1984.

[TIHW01] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. *SIGMOD Rec.*, 30(2):413–424, 2001.

[UAH76] J. D. Ullman, A. V. Aho, and D. S. Hirschberg. Bounds on the complexity of the longest common subsequence problem. *J. ACM*, 23(1):1–12, 1976.

[Val84] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[Via01] V. Vianu. A web odyssey: from Codd to XML. In *ACM Symposium on Principles of Database System*, 2001.

[Via03] V. Vianu. XML: From practice to theory. In *Simpósio Brasileiro de Banco de Dados (SBBD)*, 2003.

[W3C00] W3C. World Wide Web Consortium. In *www.w3c.org*, 2000.

[WF74] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[WHA+00] L. Wood, A. Le Hors, V. Apparao, S. Byrne, M. Champion, S. Issacs, I. Jacobs, G. Nicol, J. Robie, R. Sutor, and C. Wilson. *Document Object Model (DOM) Level 1 Specification*. W3C Recommendation, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/, 2000.

[Woo87] D. Wood. *Theory of Computation: A Primer*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[ZPC97] D. Ziadi, J.-L. Ponty, and J.-M. Champarnaud. Passage d'une expression rationnelle à un automate fini non-déterministe. *Bulletin of the Belgian Mathematical Society*, pages 177–203, 1997.

Annexe A

Algorithme A.0.1 - Conversion d'une DTD en un ENFTA

01. `Entrée:` Une DTD d
02. `Sortie:` l'ENFTA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$
03. L'ensemble d'automates de Glushkov SM_E, pour chaque E dans Δ
04. La table $T[att\text{-}name, att\text{-}kind, att\text{-}status]$
05. $S_{compulsory} = S_{optional} = SM_E = Q = \Sigma = \emptyset$;
06. $S =< S_{compulsory}, S_{optional} >$;
07. `for each element declaration` <!ELEMENT ele $\mathcal{C}$> `in` d `do {`
00. $\Sigma = \Sigma \cup \{ele\}$
10. $Q = Q \cup \{q_{ele}\}$
11. `switch` $(\mathcal{C})$
12. `{`
13. `case` $PCDATA$: $E = q_{data}$
14. `Let` δ `be` $ele,\ S,\ E \rightarrow q_{ele}$
15. $//*$ Glushkov(E) returns a Glushkov automaton from E
16. $M_E = Glushkov(E \cdot \#)$
17. $SM_E = SM_E \cup \{M_E\}$
18. $\Sigma = \Sigma \cup \{data\}$
18. $Q = Q \cup \{q_{data}\}$
19. $\Delta = \Delta \cup \{data, < \emptyset, \emptyset >, \emptyset \rightarrow q_{data}\}$
20. `case` $regExp$: `Let E be` $regExp$ `with all` $a \in regExp$ `replaced by` q_a
21. `Let` δ `be` $ele,\ S,\ E \rightarrow q_{ele}$
22. $M_E = Glushkov(E \cdot \#)$
17. $SM_E = SM_E \cup \{M_E\}$
24. `case` $EMPTY$: $E = \emptyset$
25. `Let` δ `be` $ele,\ S,\ E \rightarrow q_{ele}$
26. `}`
27. $\Delta = \Delta \cup \{\delta\}$
28. `}`
29. `for each attribute declaration` <!ATTLIST ele attSet> `in` d `do {`
31. `for each` $tuple\ \nu[att\text{-}name, att\text{-}status, att\text{-}kind]$ `in` $attSet$ `do {`
32. $attName = \nu(att\text{-}name)$
33. $\Sigma = \Sigma \cup \{attName\}$
34. $Q = Q \cup \{q_{attName}\}$
35. $\Delta = \Delta \cup \{attName, < \emptyset, \emptyset >, q_{data} \rightarrow q_{attName}\}$
36. `if` $(\nu(att\text{-}status) = REQUIRED)$ `{`
37. `for each` $\delta : ele,\ S,\ E \rightarrow q_{ele}$ `already in` Δ `do {`
38. $S_{compulsory} = S_{compulsory} \cup \{q_{attName}\}$ `}`

39. `}`
40. `else {`
41. `for each` $\delta : ele,\ S,\ E \rightarrow q_{ele}$ `already in` Δ `do {`
42. $S_{optional} = S_{optional} \cup \{q_{attName}\}$ `}`
43. `}`
44. `Insert Into T values <E:eleName, A:attName, K:`$\nu(att\text{-}kind)$`>`
45. `}`
46. `}`
47. `Let` $Q_f = \{q_{firstEle}\}$

L'exemple suivant montre le résultat de l'exécution de l'algorithme A.0.1 sur la DTD de la figure 7.2.

Exemple A.0.1 En prenant la DTD de la figure 7.2 comme entrée de l'algorithme *Conversion d'une DTD en un ENFTA*, nous aurons les sorties suivantes:

1. L'automate d'arbre $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, où:
   $Q = \{q_{Annee}, q_{CId}, q_{Chercheur}, q_{IDAut}, q_{Laboratoire}, q_{Nom}, q_{Publication}, q_{Revue}, q_{Sujet}, q_{TArticle}, q_{Titre}, q_{Universite}, q_{data}\}$
   $\Sigma = \{Annee, CId, Chercheur, IDAut, Laboratoire, Nom, Publication, Revue, Sujet, TArticle, Titre, Universite, data\}$
   $Q_f = \{q_{Universite}\}$
   Δ est présenté dans la table suivante:

	a	$<S_{compulsory}, S_{optional}>$	**E**	q_a
1	$Annee$	$\emptyset$,$\emptyset$	q_{data}	q_{Annee}
2	CId	$\emptyset$,$\emptyset$	q_{data}	q_{CId}
3	$Chercheur$	$\{q_{CId}\}$,$\emptyset$	$q_{Nom}\ q_{Titre}$	$q_{Chercheur}$
4	$IDAuts$	$\emptyset$,$\emptyset$	q_{data}	q_{IDAuts}
5	$Laboratoire$	$\emptyset$,$\emptyset$	$q_{Nom}\ {q_{Chercheur}}^*\ {q_{Publication}}^*$	$q_{Laboratoire}$
6	Nom	$\emptyset$,$\emptyset$	q_{data}	q_{Nom}
7	$Publication$	$\emptyset$,$\emptyset$	$q_{Sujet}(q_{Annee}\ {q_{Revue}}^+)^*$	$q_{Publication}$
8	$Revue$	$\emptyset$,$\emptyset$	$q_{Nom}\ q_{TArticle}$	q_{Revue}
9	$Sujet$	$\emptyset$,$\emptyset$	q_{data}	q_{Sujet}
10	$TArticle$	$\{q_{IDAuts}\}$,$\emptyset$	q_{data}	$q_{TArticle}$
11	$Titre$	$\emptyset$,$\emptyset$	q_{data}	q_{Titre}
12	$Universite$	$\emptyset$,$\emptyset$	${q_{Laboratoire}}^*$	$q_{Universite}$
13	$data$	$\emptyset$,$\emptyset$	q_{data}	q_{data}

2. L'ensemble d'automates de Glushkov SM_E:
   M_{Annee}=({0,1,2},{q_{data},#},{(0,q_{data})=1,(1,#)=2},0,{2})
   $M_{Chercheur}$=({0,1,2,3},{q_{Nom},q_{Titre},#},{(0,q_{Nom})=1,(1,q_{Titre})=2,(2,#)=3},0,{3})
   M_{CId}=({0,1,2},{q_{data},#},{(0,q_{data})=1,(1,#)=2},0,{2})
   M_{IDAuts}=({0,1,2},{q_{data},#},{(0,q_{data})=1,(1,#)=2},0,{2})
   $M_{Laboratoire}$=({0,1,2,3,4},{q_{Nom},$q_{Chercheur}$,$q_{Publication}$,#},{(0,q_{Nom})=1,(1,$q_{Chercheur}$)=2, (1,#)=4,(2,$q_{Chercheur}$)=2,(2,$q_{Publication}$)=3,(3,$q_{Publication}$)=3, (3,$q_{Publication}$)=4,(2,#)=4},0,{3})
   M_{Nom}=({0,1,2},{q_{data},#},{(0,q_{data})=1,(1,#)=2},0,{2})
   $M_{Publication}$=({0,1,2,3,4},{q_{Sujet},q_{Annee},q_{Revue},#},{(0,q_{Sujet})=1,(1,q_{Annee})=2,(1,#)=4, (2,q_{Revue})=3,(3,q_{Annee})=2,(3,q_{Revue})=3,(3,#)=4},0,{4})
   M_{Revue}=({0,1,2,3},{q_{Nom},$q_{TArticle}$,#},{(0,q_{Nom})=1,(1,$q_{TArticle}$)=2,(2,#)=3},0,{3})
   M_{Sujet}=({0,1,2},{q_{data},#},{(0,q_{data})=1,(1,#)=2},0,{2})

$M_{TArticle}$=({0,1,2},{q_{data},#},{(0,q_{data})=1,(1,#)=2},0,{2})
M_{Titre}=({0,1,2},{q_{data},#},{(0,q_{data})=1,(1,#)=2},0,{2})
$M_{Universite}$=({0,1,2},{$q_{Laboratoire}$,#},{(0,$q_{Laboratoire}$)=1,
(0,#,2),(1,$q_{Laboratoire}$)=1,(1,#)=2},0,{2})
M_{data}=({0,1,2},{q_{data},#},{(0,q_{data})=1,(1,#)=2},0,{2})

3. La table de propriétes des attributs T:

Nom d'élément	Nom d'attribut	Type
$Chercheur$	CId	ID
$TArticle$	$IDAuts$	$IDREFS$

Annexe B

Théorème 7.2.2 *Étant donné une DTD d et l'automate d'arbre $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ construit selon la définition 7.1.1, le langage généré par d est équivalent au langage reconnu par $\mathcal{A}$, i.e., $L(d) = L(\mathcal{A})$.*

Preuve: Reprenons la définition 3.3.3.
Soit d une DTD avec l'alphabet $\Sigma = \Sigma_{ele} \cup \Sigma_{att} \cup \{data\}$. La grammaire d'arbre régulière locale $D = (N, \Sigma_{ele} \cup \{data\}, \Sigma_{att}, P, R_{comp}, R_{opt}, firstEle)$ qui défini d est :

- $N = \Sigma$.
- P est une application qui fait correspondre les éléments de N à leurs définitions, c'est-à-dire, $P(\alpha)$ a la forme $P(\alpha) = e(E|\epsilon)$
  où $\alpha \in N$, $e \in \Sigma_{ele} \cup \{data\}$, E est une expression régulière sur N et ϵ représente le mot vide (remarquer que dans les grammaires locales $\alpha = e$).
- R_{comp} est une application partielle qui fait correspondre $\Sigma_{ele} \cup \{data\}$ à $\mathcal{P}(\Sigma_{att})$ (*i.e.*, l'ensemble des parties de Σ_{att}); si $a \in R_{comp}(e)$ alors a est un attribut obligatoire défini pour e.
- R_{opt} est une application partielle qui fait correspondre $\Sigma_{ele} \cup \{data\}$ à $\mathcal{P}(\Sigma_{att})$ (*i.e.*, l'ensemble des parties de Σ_{att}); si $a \in R_{opt}(e)$ alors a est un attribut optionnel défini pour e.
- $firstEle \in N$ est l'élément racine. □

On note $R(e) = R_{comp}(e) \cup R_{opt}(e)$. Étant donné d et un arbre t étiqueté en N, l'arbre t est un arbre dérivé de d si et seulement si:

g.1 La racine de t a pour étiquette $firstEle$.

g.2 Pour tout nœud dans une position interne p étiqueté $a \in \Sigma$, ayant des fils aux positions $p0 \dots p(n-1)$:

a. Il existe i, $0 \leq i \leq n$, tel que pour $j \in [0 \dots i-1]$ chaque pj a pour étiquette a_j et $\{a_0 \dots a_{(i-1)}\} \subseteq R(a)$, plus précisément :
(i) $R_{comp}(a) \subseteq \{a_0 \dots a_{(i-1)}\}$ et
(ii) $\{a_0 \dots a_{(i-1)}\} \setminus R_{comp}(a) \subseteq R_{opt}(a)$

b. Pour tout $j \in [i \dots n-1]$ chaque pj a pour étiquette e_j et $e_i \dots e_{n-1} \in L(P(a))$

g.3 Tout nœud feuille a pour étiquette $data$, ou sinon $a \in \Sigma_{ele}$ tel que $P(a) = \epsilon$ ou $\epsilon \in L(P(a))$, et $R_{comp}(a) = \emptyset$.

D'abord, nous rappelons que $A = (Q, \Sigma, Q_f, \Delta)$ est construit à partir de d en respectant la définition 7.2.1. Ensuite, nous rappelons le morphisme h qui fait correspondre les éléments dans Σ à ceux dans Q, tel que pour chaque q_a dans Q, $h(q_a) = a$. Pour terminer, comme nous supposons que t est un arbre dérivé de d, nous savons que toutes les propriétés de **g.1** à **g.3** sont respectées.

Première partie : Nous prouvons que $L(d) \subseteq L(\mathcal{A})$ en supposant que t est un arbre dérivé de d et en montrant que $\mathcal{A}$ accepte t. Pour prouver que $\mathcal{A}$ accepte t dérivé de d nous montrons que toutes les conditions (items) de la définition 7.1.2 sont respectées:

- Item 1:
  À partir des items **g.1** à **g.3** nous pouvons voir que pour chaque position $p \in dom(t)$, $t(p) \in N$, l'item 1 est donc respecté.
- Item 2:
  Premièrement, nous rappelons que, à partir de la définition 7.2.1, du morphisme h et de la règle de transition $a, < S_{compulsory}, S_{optional} >, E \rightarrow q_a$:
  $S_{compulsory} = \{h(l) \mid l \in R_{comp}(a)\}$,
  $S_{optional} = \{h(l) \mid l \in R_{opt}(a)$ et $h(l) \notin S_{compulsory}\}$ et pour tous a, a_1, a_2 de N, H est défini par :
  $$E = H(P(a)) \text{ où } \begin{cases} H(\epsilon) = \epsilon \\ H(\emptyset) = \emptyset \\ H(a) = q_a \\ H(a_1 a_2) = q_{a_1} q_{a_2} \\ H(a_1 \mid a_2) = q_{a_1} \mid q_{a_2} \\ H(a^*) = q_a^* \\ H(a^+) = q_a^+ \end{cases}$$
  Deuxièmement, les applications $P(a)$ et $R(a)$ sont traduites par la règle $a, < S_{compulsory}, S_{optional} >, E \rightarrow q_a$ dans Δ. Donc l'item 2 est respecté.
- Item 3:
  L'item **g.2** montre que les fils $p_0 \dots p_{(n-1)}$ d'une position $p \in dom(t)$ étiquetée $a \in N$ sont construits en utilisant les applications P et $\mathcal{R}$ de la grammaire de d. Alors, les fils de p sont soit des membres d'une séquence (des éléments) engendrés par l'application P (qui par le morphisme h, correspond à l'expression régulière E), soit des membres d'un ensemble (des attributs) engendrés par R (qui par le morphisme h, correspond aux ensembles $< S_{compulsory} >$ et $< S_{optional} >$). Et ainsi, l'item 3 est respecté.
- Les items 4, 5 and 6:
  Un nœud dans la position $p \in dom(t)$ peut être:

  1. Une feuille (item **g.3**), *i.e.*, (*i*) un nœud étiqueté *data*, (*ii*) un nœud qui représente un élément dont le modèle de contenu est $EMPTY$ sans attributs obligatoires (*i.e.*, $P(a) = \epsilon$ et $R_{comp}(a) = \emptyset$) ou (*iii*) un élément dont le modèle de contenu accepte le mot vide et qui n'a pas d'attributs obligatoires attachés, *i.e.*, $\epsilon \in P(a)$ et $R_{comp}(a) = \emptyset$, respectivement. Pour le premier cas, $\mathcal{A}$ associe l'état q_{data} au nœud. Pour le second et troisième cas $\mathcal{A}$ associe l'état q_a au nœud car le nœud de l'arbre a l'étiquette a, selon $P(a)$ et $R(a)$, respectivement.
     Dans la définition 7.2.1, la règle de transition est soit $a, < \emptyset, S_{optional} >, \emptyset \rightarrow q_a$ car $P(a) = \epsilon$ et $R_{comp} = \emptyset$ soit $a, < \emptyset, S_{optional} >, E \rightarrow q_a$ car $P(a) = h(E)$ $(\epsilon \in L(E))$ et $R_{comp} = \emptyset$, respectivement.

  2. Un nœud interne (item **g.2**) ou le nœud racine (item **g.1**): Dans ce cas il existe $P(a)$ qui construit un nœud étiqueté a dans la position p ayant des fils $p_0 \dots p_{(n-1)}$. Soit $\{a_0, \dots, a_{i-1}\}$ $(0 \leq i < n)$ des étiquettes dans les positions pj $(j < i)$ telles que $\{a_0, \dots, a_{i-1}\} \subseteq R(a)$. Soit $\{a_i, \dots, a_{n-1}\}$ des étiquettes dans les positions pj $(j < n)$ telles que $a_i \dots a_{n-1} \in P(a)$. Nous supposons que les fils $a_0 \dots a_{(n-1)}$ de a ont déjà été associés aux états $q_{a_0} \dots q_{a_{(n-1)}}$ comme hypothèse inductive.
     Par l'item 2 de la définition 7.1.2, nous savons qu'il existe une règle de transition $a, < S_{compulsory}, S_{optional} >, E \rightarrow q_a$ qui associe l'état q_a à a.

Donc, nous avons :

- Par l'hypothèse inductive, l'item 4 est respecté.
- La suite implique que l'item 5 est respecté :
  - Comme $a_i \dots a_{(n-1)}$ est construit en suivant $P(a)$, alors il existe un automate d'états finis $M_{P(a)}$ qui accepte $a_i \dots a_{(n-1)}$. Nous savons que $P(a)$ et $a_i \dots a_{(n-1)}$, en utilisant le morphisme h, correspondent à E et $q_{a_i} \dots q_{a_{(n-1)}}$, respectivement. Et ainsi, il existe un automate d'états finis M_E qui accepte $q_{a_i} \dots q_{a_{(n-1)}}$, *i.e.*, $q_{a_i} \dots q_{a_{(n-1)}} \in L(E)$.
  - Si la séquence $a_i \dots a_{(n-1)}$ est vide, alors $M_{P(a)}$ accepte le *mot vide* et, par conséquent, M_E l'accepte aussi.
- L'item 6 est respecté car:
  - Selon l'item **g.2 (a) (i)**, nous savons que $R_{comp}(a) \subseteq \{a_0, \dots, a_{(i-1)}\}$, avec $i \in [0 \dots (n-1)]$. Selon la définition 7.2.1 et le morphisme h nous avons $h(R_{comp}(a)) = S_{compulsory}$. En appliquant le morphisme h sur $\{a_0, \dots, a_{(i-1)}\}$, nous obtenons $\{q_{a_0}, \dots, q_{a_{(i-1)}}\}$. Donc, $S_{compulsory} \subseteq \{q_{a_0}, \dots, q_{a_{(i-1)}}\}$.
  - Selon l'item **g.2 (a)(ii)**, nous savons que $\{a_0, \dots, a_{(i-1)}\} \setminus R_{comp}(a) \subseteq R_{opt}(a)$. En utilisant le même raisonnement que l'item ci-dessus nous avons: $\{q_{a_0}, \dots, q_{a_{(i-1)}}\} \setminus S_{compulsory} \subseteq S_{optional}$.

  Et ainsi, nous avons prouvé que pour chaque arbre dérivé t de d il existe une exécution complète et réussie de $\mathcal{A}$ sur t, *i.e.*, $L(d) \subseteq L(\mathcal{A})$.

Deuxième partie: nous prouvons que $L(\mathcal{A}) \subseteq L(d)$.

Soit un arbre t accepté par notre automate d'arbre étendu $\mathcal{A}$.
Nous montrons qu'alors t est un arbre dérivé de la DTD d, vue comme une grammaire hors contexte étendue.

Étant donné l'arbre t et l'automate d'arbre $\mathcal{A}$, t est accepté par $\mathcal{A}$ si et seulement si chaque position p d'un nœud étiqueté a dans t peut être associé à un état q_a ($q_a \in Q$) et l'état associé à la racine $q_{firstEle}$ est dans Q_f.

- Selon sur la définition 7.2.1, nous savons que la position de la racine ϵ dans t est étiquetée $firstEle$. Si $firstEle$ étiquette ϵ dans t alors il existe une règle de transition dans $\mathcal{A}$ qui associe l'état $q_{firstEle}$ à ϵ et $q_{firstEle} \in Q_f$. Puisque $firstEle$ est l'étiquette du nœud racine de t, $r = firstEle$ dans d et ainsi, l'item **g.1** est respecté.
- Une position feuille p dans t est étiquetée par $data \in \Sigma$ ou par une étiquette $a \in \Sigma$ et, dans les deux cas, $\mathcal{A}$ associe q_{data} ou q_a en utilisant la règle de transition $data, < \emptyset, \emptyset >, \emptyset \rightarrow q_{data}$ ou $a, < \emptyset, S_{optional} >, E \rightarrow q_a$, avec le *mot vide* dans $L(E)$, respectivement. Dans d nous savons que p est construite soit par $P(a) = data$ et $\mathcal{R}(a) = \emptyset$ sur le père de p soit sur p lui-même par $P(a)$ et $R_{comp}(a) = \emptyset$ tel que $\epsilon \in L(P(a))$.
  Nous savons alors que $P(a) = data$, $R_{comp} = \emptyset$ et $\epsilon \in L(P(a))$, $\mathcal{R}(a) = \emptyset$, en utilisant le morphisme h et la définition 7.2.1, correspondent à $data, < \emptyset, \emptyset >, \emptyset \rightarrow q_{data}$ et $a, < \emptyset, S_{optional} >, E \rightarrow q_a$, respectivement. Donc, l'item **g.3** est respecté.
- Une position interne $p \in dom(t)$, *i.e.*, $children(t, p) = p_0, \dots, p_{n-1}$, étiquetée par a est associée a un état q_a si ses fils $a_0, \dots, a_n$ ont déjà été associés aux états $q_{a_0}, \dots, q_{a_{n-1}}$ et s'il existe une règle de transition $a, < S_{mandatory}, S_{optional} >, E \rightarrow q_a$, telle que $S_{mandatory} \subseteq \{q_{a_0}, \dots, q_{a_{i-1}}\}$, $\{q_{a_0}, \dots, q_{a_{i-1}}\} \setminus S_{mandatory} \subseteq S_{optional}$, avec $i \in [0 \dots n-1]$, et $q_{a_i} \dots q_{a_{n-1}} \in L(E)$ (définition 7.1.2). En appliquant d, nous savons qu'une position interne p de t étiqueté a est construite par P telle que $\epsilon \notin L(P(a))$ ou $R_{comp}(a) \neq \emptyset$. Alors, les fils de p, $a_0, \dots, a_{n-1}$ sont construits par R, avec $R_{comp}(a) \subseteq \{a_0, \dots, a_{i-1}\}$ et $\{a_0, \dots, a_{i-1}\} \setminus R_{comp}(a) \subseteq R_{opt}(a)$ ($i \in [0 \dots (n-1)]$), ou par $P(a)$, avec $a_i \dots a_{n-1} \in L(P(a))$.

En appliquant le morphisme h et la définition 7.2.1, nous avons $\{q_{a_0}, \ldots, q_{a_{i-1}}\}$ et $S_{mandatory} \subseteq \{q_{a_0}, \ldots, q_{a_{i-1}}\}$, $\{q_{a_0}, \ldots, q_{a_{i-1}}\} \setminus S_{mandatory} \subseteq S_{optional}$, et $q_{a_i} \ldots q_{a_{n-1}} \in L(E)$. Ainsi, l'item **g.2** est respecté.

– Les items **g.2 a (i)** et **g.2 a (ii)** sont respectés selon la définition du couple S de la règle de transition $a, S, E \rightarrow q_a$ de $\mathcal{A}$ (définition 7.2.1).

Et ainsi, nous avons prouvé que pour chaque t accepté par $\mathcal{A}$, t est dérivé de d, *i.e.*, $L(\mathcal{A}) \subseteq L(d)$.

Il en découle que si $\mathcal{A}$ est l'automate d'arbre construit à partir de la DTD d en suivant la définition 7.1.2, alors $L(d) = L(\mathcal{A})$. □

Annexe C

Théorème 8.5.1 *Soient G un graphe sans orbite réductible et $\mathcal{H}$ la hiérarchie des orbites de G. Soit R_i l'une des règles de réduction $\boldsymbol{R}_1$, $\boldsymbol{R}_2$ ou $\boldsymbol{R}_3$. Pour tous les nœuds s_{nl}, s_{nr} et s_{new}, les graphes G_{new} résultants de l'exécution de* `LookForGraphAlternative`*(G, $\mathcal{H}$, R_i, s_{nl}, s_{nr}, s_{new}) sont réductibles.*

Preuve: D'abord, nous savons, par la proposition **4.4** et par le théorème **5.1** dans [CZ00], que le graphe G, utilisé dans le contexte de `GREC`, est réductible.
Ainsi, nous allons démontrer que les modifications faites sur G par *LookForGraphAlternative*, donnant G_{new}, en utilisant les définitions 8.4.3 à 8.4.6, maintiennent la propriété de réductibilité.

Rappelons que, étant donné un graphe $G = (X, U)$ sans orbite, G est dit *réductible* s'il est possible de le réduire à un nœud en appliquant successivement l'une des trois règles $\mathbf{R}_1$, $\mathbf{R}_2$ et $\mathbf{R}_3$ (définition 8.3.6). Ces règles peuvent être résumées comme suit :

- $\mathbf{R}_1$: s'il existe deux nœuds x et y dans G tels que $Q^-(y) = \{x\}$ et $Q^+(x) = \{y\}$, alors l'arc (x, y) et le nœud y sont supprimés, le nœud x devient xy et tous les arcs $(y, Q^+(y))$ et $(Q^-(x), x)$ sont mis à jour ce qui donne $(xy, Q^+(y))$ et $(Q^-(x), xy)$, respectivement.
- $\mathbf{R}_2$: s'il existe deux nœuds x et y dans G tels que $Q^-(x) = Q^-(y)$ et $Q^+(x) = Q^+(y)$, alors les arcs $(Q^-(y), y)$ et $(y, Q^+(y))$, et le nœud y sont supprimés, le nœud x devient $x|y$ et tous les arcs $(Q^-(x), x)$ et $(x, Q^+(x))$ sont mis à jour ce qui donne $(Q^-(x), x|y)$ et $(x|y, Q^+(x))$, respectivement.
- $\mathbf{R}_3$: S'il existe un nœud x tel que $y \in Q^-(x) \Rightarrow Q^+(x) \subset Q^+(y)$, alors les arcs $(Q - (x), Q^+(x))$ sont supprimés, le nœud x devient $x?$ (ou x^* si x^+).

Nous allons démontrer que G_{new} est réductible. Pour cela, nous allons présenter les cas pour chaque règle/définition à appliquer. Remarquer que le graphe G passé comme paramètre à `LookForGraphAlternative` a toujours plus que 1 nœud.

$\mathbf{R}_1$: Deux nœud x et y ont été choisis pour être fusionnés par cette règle. Dans ce contexte, la définition 8.4.3 peut être appliquée par `LookForGraphAlternative` :
(i) Cas 1 : G_{new} aura, en plus, un nouveau nœud z, et les arcs (x, z) et (z, y). Dans ce contexte, d'abord la règle $\mathbf{R}_3$ peut être appliquée sur z et l'arc (x, y) est supprimé et le nœud z devient $z?$. Ensuite, la règle $\mathbf{R}_1$ peut être appliquée sur les nœuds x et z ou z et y. Comme résultat, G_{new} aura l'arc $(xz?, y)$ ou $(x, z?y)$. Le graphe G_{new} revient au graphe G et ainsi, G_{new} est réductible.
(ii) Cas 2 : deux nouveaux graphes sont construits : G^1_{new} avec le nouveau nœud z et des nouveaux arcs ajoutés, à savoir (y, z) et $(z, Q^+(y))$ et, G^2_{new} avec z et des nouveaux arcs ajoutés, à savoir (z, x) et $(Q^-(x), z)$.
G^1_{new} est réductible car la règle $\mathbf{R}_3$ peut être appliquée sur z, les arcs $(y, Q^+(y))$ sont alors supprimés, et ensuite la règle $\mathbf{R}_1$ peut être appliqué sur y et z. Ainsi, des ajouts faits par `LookForGraphAlternative` à G^1_{new}, les seuls arcs qui restent sont $(yz?, Q^+(y))$. Ces arcs représentent les arcs $(y, Q^+(y))$ de G, alors G^1_{new} est réductible.

G^2_{new} est réductible car la règle $\mathbf{R}_3$ peut être appliquée sur z, les arcs $(Q^-(x), x)$ sont alors supprimés, et ensuite la règle $\mathbf{R}_1$ peut être appliquée sur z et x. En utilisant les mêmes principes du graphe G^1_{new}, les seuls arcs qui restent sont $(Q^-(x), z?x)$ qui représentent les arcs $(Q^-(x), x)$ de G, alors G^2_{new} est aussi réductible.

$\mathbf{R}_2$: Deux nœud x et y ont été choisis pour être fusionnés par cette règle. Dans ce contexte, la définition 8.4.4 peut être appliquée par `LookForGraphAlternative` :
Cette définition a trois cas et six nouveaux graphes sont construits à partir de G. Cependant, il existe seulement trois types de graphes : $G^{t_1}_{new}$, $G^{t_2}_{new}$ et $G^{t_3}_{new}$.
$G^{t_1}_{new}$ est construit en ajoutant le nœud z et les arcs (x, z) et $(z, Q^+(x))$. Ce type de graphe est similaire au graphe G^1_{new} construit par la définition 8.4.3 ci-dessus et ainsi, en appliquant le même raisonnement, on conclut que $G^{t_1}_{new}$ est aussi réductible.
$G^{t_2}_{new}$ est construit en ajoutant le nœud z et les arcs (z, x) et $(Q^-(x), z)$. Ce type de graphe est similaire au graphe G^2_{new} construit par la définition 8.4.3 ci-dessus et ainsi, en appliquant le même raisonnement, on conclut que $G^{t_2}_{new}$ est aussi réductible.
$G^{t_3}_{new}$ est construit en ajoutant le nœud z et les arcs $(Q^-(x), z)$ et $(z, Q^+(x))$ (ou $(Q^-(y), z)$ et $(z, Q^+(y))$). Dans ce contexte, les nœuds x et z (ou y et z) seront fusionnés par la règle $\mathbf{R}_2$ ce qui donne un seul nœud $x|z$ (ou $y|z$) et ainsi, le nœud et les arcs ajoutés par `LookForGraphAlternative` sont supprimés de $G^{t_3}_{new}$. Ainsi $G^{t_3}_{new}$ a les nœuds et les arcs d'origine (du graphe G) et donc, $G^{t_3}_{new}$ est réductible.

$\mathbf{R}_3$: Un nœud x a été choisi pour devenir optionnel par cette règle. Dans ce contexte, la définition 8.4.5 peut être appliquée par `LookForGraphAlternative` :
Cette définition a un cas et trois nouveaux graphes sont construits à partir de G :
Le graphe G^1_{new} dans lequel le nœud z et les arcs $(Q^-(x), z)$ et (z, x) sont ajoutés. G^1_{new} est réductible car la règle $\mathbf{R}_3$ peut être appliquée sur z, les arcs $(Q^-(x), x)$ sont alors supprimés, et ensuite la règle $\mathbf{R}_1$ peut être appliquée sur les nœuds z et x : le nœud x est supprimé et le nœud z devient $z?x$. Après l'application de ces deux règles, G^1_{new} représente le graphe d'origine G (avec le nœud x représenté par le nœud $z?x$) et ainsi, G^1_{new} est réductible.
La construction du G^2_{new} est symétrique à la construction du graphe G^1_{new} et, en appliquant les mêmes principes, on conclut que G^2_{new} est aussi réductible.
Le graphe G^3_{new} est construit en suivant les mêmes pas que pour le graphe $G^{t_3}_{new}$ de la définition 8.4.4 ci-dessus et ainsi, G^3_{new} est aussi réductible.

Décoration d'un nœud avec $^+$: Lorsqu'un nœud x dans G représente une orbite complète, ce nœud est décoré avec l'opérateur $^+$. Avant d'appliquer la décoration sur un nœud, la définition 8.4.6 peut être appliquée par `LookForGraphAlternative` :
Cette définition a quatre cas et huit nouveaux graphes sont construits à partir de G. Cependant, comme dans le cas de la définition 8.4.4, il existe seulement trois types de graphes : $G^{t_1}_{new}$, $G^{t_2}_{new}$ et $G^{t_3}_{new}$.
$G^{t_1}_{new}$ est construit en ajoutant le nœud z et les arcs (x, z) et $(z, Q^+(x))$. Ce type de graphe est similaire au type $G^{t_1}_{new}$ de la définition 8.4.4 et donc, il est réductible.
$G^{t_2}_{new}$ est construit en ajoutant le nœud z et les arcs (z, x) et $(Q^-(x), z)$. Ce type de graphe est similaire au type $G^{t_2}_{new}$ de la définition 8.4.4 et donc, il est réductible.
$G^{t_3}_{new}$ est construit en ajoutant le nœud z et les arcs $(Q^-(x), z)$ et $(z, Q^+(x))$ (ou $(Q^-(y), z)$ et $(z, Q^+(y))$). Ce type de graphe est similaire au type $G^{t_3}_{new}$ de la définition 8.4.4 et donc, il est réductible.

Et ainsi, nous avons prouvé que les graphes G_{new}, construits par la fonction `LookForGraphAlternative`(G, $\mathcal{H}$, R_i, s_{nl}, s_{nr}, s_{new}) sont toujours réductibles. □

Annexe D

Théorème 8.5.2 *Soit E une expression régulière et $L(E)$ le langage associé à E. Étant donné $w[0:n] \in L(E)$ $(0 \leq n)$, soit $w_{ins}[0:n+1]$ (respectivement $w_{del}[0:n-1]$) un mot résultat d'une opération d'insertion (respectivement d'une opération de suppression) sur une position p (avec $0 \leq p \leq n$) tel que $w_{ins} \notin L(E)$ (respectivement $w_{del} \notin L(E)$). Soient M_E l'automate de Glushkov construit à partir de E et G le graphe obtenu à partir de M_E. Soit $(G_{wo}, \mathcal{H})$ la paire qui représente le graphe sans orbite et la hiérarchie des orbites de G, respectivement. Soit s_{nl} un état dans M_E tel que $s_{nl} = \hat{\delta}(q_0, \alpha)$ (où $\alpha = w[0:p-1]$). Soit s_{nr} un état dans M_E tel que $s_{nr} = \hat{\delta}(q_0, \alpha)$ (où $\alpha = w[0:p+1]$). Soit s_{new} un état qui n'existe pas dans Q pour les insertions ou $s_{new} = \hat{\delta}(q_0, \alpha)$ tel que $\alpha = w[0:p]$ pour les suppressions.*
L'exécution de `GREC` $(G_{wo}, \mathcal{H}, s_{nl}, s_{nr}, s_{new})$ *retourne un ensemble fini et non vide d'expressions régulières candidates $\{E_1, \ldots, E_m\}$. De plus, pour tout $E_i \in \{E_1, \ldots, E_m\}$, nous avons $L(E) \cup \{w'\} \subset L(E_i)$ et $\mathcal{D}(E, E_i) = 1$ pour les insertions et $\mathcal{D}(E, E_i) = 0$ pour les suppressions.*

Preuve: Pour prouver que l'ensemble d'expressions régulières candidates construit par `GREC` est fini et non vide, nous rappelons que tous les candidats sont construits en respectant les conditions et les cas présentés dans les définitions 8.4.3 à 8.4.7. Ainsi, les points suivants montrent que l'ensemble construit par `GREC` est fini et non vide.

- Le nombre de graphes proposés pour chaque définition est fini et, par définition, `GREC` termine car, à un moment donné, le graphe d'origine est réduit à un nœud et c'est la condition de fin de `GREC`. Donc, le nombre de candidats est fini.
- Dans le cas d'une insertion, l'ensemble de candidates est non vide car les définitions 8.4.3 à 8.4.6 couvrent tous les cas des règles de réduction. Comme s_{nl} et s_{nr} sont des nœuds qui appartiennent au graph à réduire, l'une des règles de réduction va les fusionner et ainsi, l'un (ou plus) de cas des défintions 8.4.3 à 8.4.6 vont être satisfait et des nouvelles expressions régulières seront proposées.
- Dans le cas d'une suppression, la définition 8.4.7 construit un nouveau graphe en rendant optionnel un nœud s_{new} qui représente un état obligatoire dans un contexte, en ajoutant des arcs de tous les prédécesseurs de s_{new} vers tous les successeurs de s_{new}. Donc, pour les suppressions, `GREC` construit exactement un nouveau candidat et ainsi, l'ensemble de candidats est non vide.

Dans la suite, nous prouvons la correction de notre méthode, *i.e.*, étant donné l'expression régulière d'origine E, $w \in L(E)$ et w' construit à partir w en insérant ou en supprimant un symbole, l'ensemble d'expressions régulières $S_{E'}$ construit par `GREC` est tel que, pour toute $E_i \in S_{E'}$, $L(E) \cup \{w'\} \subseteq L(E_i)$.

Dans le cadre d'une insertion, nous savons que s_{new} est inséré pendant le processus de réduction du graphe G_{wo}. Les cas d'insertion de s_{new} sont montrés dans les définitions 8.4.3 à 8.4.6. En analysant ces définitions, on voit que elles ajoutent des arcs mais n'en suppriment pas. Donc,

tout mot de $L(E)$ est aussi dans $L(E_i)$.

Dans le cadre d'une suppression, nous savons que la définition 8.4.7 rend le s_{new} optionnel pendant le processus de réduction du graphe G_{wo}.

Alors, si M_{E_i} est construit à partir de M_E en suivant les cas présentés ci-dessus, M_{E_i} accepte tous les mots acceptés par M_E. En effet, s_{new} est soit inséré comme un état optionnel, soit comme un état disjoint, soit il est transformé en optionnel. Comme M_{E_i} est utilisé pour accepter le langage décrit par E_i, nous concluons que $L(E) \cup \{w'\} \subset L(E_i)$.

Nous prouvons que la distance entre une candidate E' et l'expression régulière d'origine E est égal à 1 en se servant de la définition 8.4.1. Dans un graphe de Glushkov, chaque nœud (état) correspond à un symbole dans l'expression régulière d'origine (hormis l'état initial), si `GREC` insère un nouvel état dans un graphe de Glushkov G et si le nouveau graphe G' est encore réductible (théorème 8.5.1), alors la transformation de G' en une expression régulière E' produit une expression régulière avec un symbole en plus par rapport à l'expression régulière produite par G, et ainsi, $\mathcal{D}(E', E) = 1$. En appliquant le même raisonnement dans le cadre de suppression, `GREC`, lors de la construction du nouveau graphe G' à partir de G, ne fait aucune insertion ou suppression de nœud. Ainsi, si le nouveau graphe a les mêmes nœuds que G, alors l'expression régulière construite à partir de G' a le même nombre de symboles que l'expression régulière construite à partir de G', *i.e.*, $\mathcal{D}(E', E) = 0$.

Et ainsi, nous avons prouvé prouvons que l'ensemble d'expressions régulières $S_{E'}$ construit par `GREC` est fini et non vide. De plus, le langage de chaque candidat inclut le langage de l'expression régulière d'origine et le nouveau mot, et la distante entre les candidats dans $S_{E'}$ et l'expression régulière d'origine E est 1 pour les insertions et 0 pour les suppressions. □

Annexe E

Théorème 9.4.2 *Soient E une expression régulière et $L(E)$ le langage associé à E. Étant donné $w[0:n] \in L(E)$ $(0 \leq n)$, soit w' un mot résultat d'une suite d'opérations de mises à jour sur des positions p (avec $0 \leq p \leq n$) de w tel que $w' \notin L(E)$. Soient M_E l'automate de Glushkov construit à partir de E et G le graphe obtenu à partir de M_E. Soit $(G_{wo}, \mathcal{H})$ la paire qui représente le graphe sans orbite et la hiérarchie des orbites de G, respectivement. Soient $RTStates$ la relation (non-vide) qui stocke les triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ et $STrans$ l'ensemble des arcs (a, e) qui restreint la construction des candidats. Soit k le nombre de triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ dans $RTStates$ tel que $s_{new} \neq null$.*
L'exécution de `GREC-e` $(G_{wo}, \mathcal{H}, RTStates, STrans)$ *retourne un ensemble fini et non vide d'expressions régulières candidates $\{E_1, \ldots, E_m\}$. De plus, pour tout $E_i \in \{E_1, \ldots, E_m\}$, nous avons $L(E) \cup \{w'\} \subset L(E_i)$ et $\mathcal{D}(E, E_i) \leq k$.*

Preuve: La preuve de ce théorème suit la preuve du théorème 8.5.2.
Pour la preuve que l'ensemble construit par `GREC-e` est fini, non-vide et correct, on se sert de la démonstration du théorème 8.5.2.
On ajoute, pour la correction, le fait que plusieurs triplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ sont utilisés pour construire un graphe et il se peut que l'une des définitions appliquées sur un triplet soit contradictoire avec une autre définition appliquée sur un autre triplet. On part de l'affirmation que toutes les candidates E_i doivent respecter : $L(E) \cup \{w'\} \subset L(E_i)$, en d'autres termes, le langage décrit par E_i doit inclure le langage décrit par l'expression régulière d'origine E et le mot w' qui n'appartient pas au langage de E.
Pour garantir que toutes les expression régulières candidates E_i décrivent le mot w' il est nécessaire de vérifier si les automates construits à partir de E_i acceptent le mot w'. L'algorithme 9.2.3 construit la relation $RTStates$ et pour chaque triplet $\langle s_{nl}, s_{nr}, s_{new}\rangle$ construit tel que $s_{new} \neq null$, une paire (a, e) est aussi construite (dans $STrans$) qui obligera les automates des candidats construits à avoir une transition $\delta(a, e) \neq \perp$.
Ainsi, si pour tous les s_{new} insérés dans un graphe, la fonction `LookForGraphAlternative-e` vérifie si la paire (a, e) est respectée alors les automates construits à partir de candidats auront la transition correspondante et donc, le mot w' est accepté par l'automate. Pour vérifier si $L(E) \subset L(E_i)$, on utilise le même raisonnement que pour le théorème 8.5.2 : les définitions ajoutent des arcs mais n'en suppriment pas, donc tout mot de $L(E)$ est aussi dans $L(E_i)$.

Dans la suite, nous prouvons que la distance entre une candidate E' et l'expression régulière E est inférieure ou égal à k en se servant de la définition 8.4.1. Dans un graphe de Glushkov, chaque nœud (état) correspond à un symbole dans l'expression régulière d'origine (hormis l'état initial).
Si `GREC-e` insère k nouveaux états dans un graphe de Glushkov G et si le nouveau graphe G' est encore réductible (théorème 9.4.1), alors la transformation de G' en une expression régulière E' produit une expression régulière avec k symboles en plus par rapport à l'expression régulière

produite par G, et ainsi, $\mathcal{D}(E', E) = k$.
En appliquant le même raisonnement dans le cadre de suppression, `GREC-e`, lors de la construction du nouveau graphe G' à partir de G, ne fait aucune insertion ou suppression de nœud. Le nouveau graphe a les mêmes nœuds que G, alors l'expression régulière construite à partir de G' a le même nombre de symboles que l'expression régulière construite à partir de G', *i.e.*, $\mathcal{D}(E', E) = 0$. Ainsi, les candidats proposés par `GREC-e` ont soit 0 nouveaux symbole (s'il n'existe que des suppressions) soit k nouveaux symboles, alors la distance entre une candidate E' et l'expression régulière d'origine est inférieure ou égal à k, *i.e.*, $\mathcal{D}(E, E') \leq k$.

Et ainsi, nous avons prouvé que l'ensemble d'expressions régulières $S_{E'}$ construit par `GREC-e` est fini et non vide. De plus, le langage de chaque candidat inclut le langage de l'expression régulière d'origine et le nouveau mot, et la distante entre les candidats dans $S_{E'}$ et l'expression régulière d'origine E est inférieure ou égal à k. □

Résumé

Nous proposons une méthode pour aider les administrateurs des applications XML dans la tâche de faire évoluer des schémas en préservant la cohérence de la base de données sans la modifier. L'utilisateur donne au système ce qu'il souhaite comme nouveau document devant être accepté par le schéma. À partir de ce document, le système construit des schémas candidats, qui d'une part préservent la validité de la base de documents et, d'autre part augmentent la classe de documents acceptée par le schéma. L'approche est implantée par un algorithme appelé `GREC`. Cet algorithme utilise l'automate d'arbre $\mathcal{A}$ qui accepte le langage défini par le schéma pour trouver les informations nécessaires à la modification. Plus précisément, il utilise les expressions régulières des règles de transitions de $\mathcal{A}$ pour proposer les candidats. Ainsi, les modifications sont faites sur les graphes qui représentent les automates d'états finis construits à partir des expressions régulières concernées. Les expressions régulières engendrées par `GREC` représentent des schémas présentés à l'utilisateur afin qu'il choisisse le plus adapté à la sémantique de son application.

Mot-clés : XML, DTD, évolution de schémas, automates d'arbres, expressions régulières, automates de Glushkov.

Abstract

We propose an approach to support administrators of XML applications, which allows XML schema evolution while preserving the consistency of existing data without any need of their modification. The user provides the system with a new document D' (possibly invalid with respect to the present schema) resulting from updating a valid document D, that triggers the evolution of the schema. From D', the system builds candidate schemas that can replace the original one and that preserve the data consistency. The system uses the associated tree automaton to find the positions in the schema that should be updated. The schema evolution is implemented by an algorithm (named `GREC`) that performs changes on the graph of a finite state automaton and that generates regular expressions for the modified graphs. Each regular expression proposed by `GREC` is given as a choice of schema to the administrator.

Keywords : XML, DTD, schema evolution, tree automata, regular expression, Glushkov automata.