View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by HAL Université de Tours

Aspects dynamiques de XML et spécification des interfaces de services web avec PEWS

Mirian Halfeld Ferrari Alves

To cite this version:

Mirian Halfeld Ferrari Alves. Aspects dynamiques de XML et spécification des interfaces de services web avec PEWS. Informatique [cs]. Université François Rabelais - Tours, 2007. <tel-00271099>

HAL Id: tel-00271099

https://tel.archives-ouvertes.fr/tel-00271099

Submitted on 8 Apr 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université François Rabelais de Tours

Année Universitaire : 2007-2008

Habilitation à Diriger des Recherches

Discipline : Informatique

présentée et soutenue publiquement par

Mírian Halfeld Ferrari Alves

le 30 novembre 2007

Aspects dynamiques de XML et spécification des interfaces de services web avec PEWS

Jury

Bernd Amann	Professeur	Université Pierre et Marie Curie	Rapporteur
Véronique Benzaken	Professeur	Université de Paris XI	Président
Giorgio Ghelli	Professeur	Università di Pisa	Rapporteur
Arnaud Giacometti	Professeur	Université François Rabelais de Tours	
Françoise Gire	Professeur	Université de Paris I	Rapporteur
Dominique Laurent	Professeur	Université de Cergy-Pontoise	
Sébastien Limet	Professeur	Université d'Orléans	
Sophie Tison	Professeur	Université de Lille 1	

Le doute est désagréable, mais la certitude est ridicule

(Voltaire)

À mon Bernard.

R E M E R C I E M E N T S

Tout d'abord je voudrais remercier très sincèrement les membres du jury. J'ai été effectivement comblée d'avoir un jury très compétant et exigeant tout en étant très *sympa* et chaleureux.

Je remercie les rapporteurs. Francoise Gire pour sa lecture attentive, ses remarques très pertinentes qui ont apporté beaucoup de précisions à mon travail. Je lui suis particulièrement reconnaissante de tout l'encouragement qu'elle a toujours su m'apporter. Giorgio Ghelli qui a accepté d'être mon rapporteur[1]. Je suis très touchée du soin avec lequel il a lu mes travaux. Ses commentaires très précis m'ont permis d'envisager différentes directions dans mes travaux. Je remercie Bernd Amman d'avoir accepté d'être mon rapporteur et pour ses suggestions et idées qui ouvrent la porte à différentes perspectives de recherche.

Je tiens à remercier *très chaleureusement* Véronique Benzaken, non seulement d'avoir accepté de présider mon jury, mais aussi par son soutien à mon travail. Je suis extrêmement touchée par la gentillesse avec laquelle elle m'a encouragée et pour la confiance qu'elle a eue en moi.

Je remercie également Sophie Tison et Sébastien Limet, pour leur immense sympathie et pour tout l'intérêt qu'ils ont porté à mes travaux, ouvrant des possibilités à de futures collaborations. Merci à Dominique Laurent pour sa confiance, et aussi pour des années de complicités et d'échanges d'opinions diverses qui m'ont toujours été très utiles. Je remercie Arnaud Giacometti pour sa gentillesse et pour son grand soutien comme collègue de travail à l'Université de Tours ainsi que pour des suggestions très intéressantes lors de ma soutenance.

Je voudrais remercier les membres du LI, spécialement les membres de l'équipe BdTln, et tous mes collègues de l'antenne de Blois qui avec compétence et solidarité m'ont beaucoup encouragée dans mon métier d'enseignant chercheur. Je remercie tous les chercheurs avec qui j'ai eu l'honneur de travailler et, en particulier ...

Je dis un énorme merci à Béatrice Bouchou, avec qui j'ai commencé mes recherches dans la domaine de XML. Le travail d'équipe que nous avons réalisé ensemble (et que, j'espère, nous poursuivrons) a toujours été très motivant. Je la remercie aussi pour ses encouragements pendant toutes ces années. Martin Musicante est un ami très spécial dont la collaboration a été essentielle tout au long du chemin qui m'a mené à mon HDR. Je le remercie aussi pour son optimisme, sa bonne humeur et son soutien. Je remercie Agata Savary pour son grand appui et la riche collaboration que nous avons entamée (en espérant que d'autres s'ensuivront).

Merci à "mes" quatre premiers doctorants ! Denio Duarte, Ahmed Cheriat, Maria Adriana de Lima (pour les mots d'encouragement et pour sa détermination) et Cheikh Ba (pour sa grande gentillesse, bonne humeur, sa disponibilité et son soutien). Les nombreuses et animées discussions scientifiques que j'ai eu avec tous les quatre ont été moteur de mes recherches.

Les voyages entre Orléans-Blois finissent par créer des liens. J'ai été très touchée par le soutien de mes nombreux collègues de train en particulier par la présence de Brigitte et Jean-Paul à ma soutenance.

Je suis très reconnaissante à mes éternels supporters. Mes parents Dalva et Amaury, qui même loin savent me transmettre du courage et de l'amour, mes beaux parents, Nicole et François pour leur exemple et leur soutien de toujours et ma soeur, Marta. J'ai la chance de pouvoir compter sur des amis loin du regard qui, dans des instants différents savent m'encourager. Je pense en particulier à Martinha (ma grande et *super* amie) et à Eugenie.

Et... *last but not least*, je remercie du fond de coeur mes deux amours Pierre et Bernard pour leur amour, encouragement, patience, compréhension ... et toutes les belles couleurs de la vie.

[1] Je tiens à préciser que le voyage de Giorgio Ghelli a été financé par les fonds de recherche de Véronique Benzaken. Merci !

Résumé

Nous nous intéressons par le problème de la sémantique des mises à jour et de la cohérence des bases de données dans différents contextes comme les documents XML et les services web. En effet, des difficultés particulières sont à prévoir lors de la mise à jour d'une base ayant des contraintes à respecter, car, des données originalement cohérentes par rapport aux contraintes peuvent devenir incohérentes suite aux mises à jour.

Dans une première partie de notre travail, nous considérons la mise à jour et le maintien de la cohérence d'une base de données XML par rapport au type (ou schéma) ainsi que par rapport aux contraintes d'intégrité. Nous abordons ce problème de différentes manières. Tout d'abord, nous proposons une procédure de validation incrémentale par rapport aux contraintes, évitant de revalider les parties du document qui n'ont pas été touchées par les mises à jour. Cette approche traite aussi bien le cas des contraintes de schéma que le cas des contraintes d'intégrité. Dans ce cadre, les listes de mises à jour qui violent la validité sont rejetées.

Quand la validation incrémentale échoue, c'est-à-dire, quand une mise à jour viole le type, deux propositions de traitement sont faites : (A) Une routine de correction est activée pour adapter le document XML au type tout en prenant en compte la mise à jour. La mise à jour a donc une priorité par rapport aux données déjà stockées. (B) Une routine propose une adaptation du type du document, de façon à accepter le document mis à jour en préservant la validité des autres documents originalement valides et non soumis à la mise à jour. Dans ce cas, la mise à jour est prioritaire et les contraintes peuvent être modifiées.

Une deuxième partie du travail considère la construction d'une plate-forme d'aide à la spécification, à l'implémentation et à la manipulation de services. L'idée de pouvoir spécifier et modifier des compositions de services nous a amené à la définition du langage PEWS (*Path Expressions for Web Services*), ayant une sémantique formelle bien définie et permettant la spécification du comportement des interfaces des services web simples ou composés. Pour pouvoir tester statiquement des propriétés liées à la composition des services, nous proposons l'utilisation de la théorie des traces et les graphes de dépendances.

Table des matières

Chapitre 1

Introduction

Les opérations de mises à jour sont fondamentales dans un système de bases de données, permettant des changements sur la base. Des difficultés particulières sont à prévoir lors de la mise à jour d'une base ayant des contraintes à respecter, car, des données originalement cohérentes par rapport aux contraintes peuvent devenir incohérentes suite aux mises à jour. Ce problème a été largement discuté dans le cadre du modèle relationnel et différentes solutions sont envisageables.

Ainsi, le problème de la mise à jour d'une base de données revient au problème plus général de mettre à jour une théorie logique (voir, par exemple, [FUV83, Win90]). La question qui se pose alors est : étant donnée une théorie T du premier ordre et considérant que de nouvelles informations sur T arrivent, comment trouver une nouvelle théorie T' prenant en compte ces nouvelles informations. Malheureusement, en général, la logique classique n'est pas assez puissante pour exprimer la sémantique des mises à jour et il n'existe pas une définition précise d'une sémantique de mise à jour indépendante de l'application. Plusieurs sémantiques ont donc été proposées.

Dans une approche traditionnelle, les mises à jour capables de violer la cohérence de la base par rapport aux contraintes sont considérées comme illégales et sont refusées. Néanmoins, les données contenues dans des mises à jour sont les informations les plus récentes qu'un utilisateur peut fournir et peuvent, dans certaines circonstances, être considérées comme prioritaires par rapport aux informations déjà stockées dans la base. Il devient alors possible de changer les informations stockées, pour pouvoir accepter la mise à jour tout en gardant la cohérence de la base. Il reste encore à savoir quel type d'information nous pouvons changer, en établissant, par exemple, des droits concernant les changements sur les données et les changements sur les contraintes elles mêmes ([FUV83]).

Ces questions, concernant les différentes approches de la mise à jour des bases de données, étaient déjà considérées dans ma thèse de doctorat ([Hal96])[1] où j'ai travaillé sur les mises à jour des bases de données déductives. Les mises à jour devaient être traitées en préservant la cohérence de la base de données tout en donnant la priorité aux nouvelles informations, c'est-à-dire, en supprimant des faits déjà stockés pour permettre l'insertion des nouveaux faits.

Dans ce cadre, nous avons proposé une approche de base de données Datalog avec des règles de mise à jour (capables de décrire les effets de bord des mises à jour) et règles de requêtes (règles classiques Datalog^{neg} permettant de déduire des faits à partir de ceux stockés explicitement dans la base). La sémantique de la base de données était fondée sur une logique à trois valeurs (un fait pouvant être vrai, faux ou inconnu) et était calculée en considérant que les règles de mise à jour avaient priorité sur les règles de requête : les règles de mise à jour pouvant engendrer des *exceptions* aux règles de requête. Une mise à jour n'était jamais refusée, mais elle déclenchait des règles de mise à jour afin de déterminer les changements nécessaires pour que la mise à jour soit effectuée ([HLS98]).

Dans la suite de mes recherches j'ai continué à m'intéresser au problème de la sémantique des mises à jour et de la cohérence des bases de données dans des cadres très différents comme les bases de données temporelles, les documents XML et, plus récemment, les services web.

[1] Thèse soutenue à l'Université de Paris Sud (XI), sous la direction de Nicolas Spyratos et le co-encadrement de Dominique Laurent.

Du point de vue des bases de données, XML propose un modèle hiérarchique, où les données sont représentées sous la forme d'un arbre avec des nœuds étiquetés et fait ainsi partie des modèles de données semi-structurées ([ABS00]). La flexibilité de XML a ouvert différentes voies pour la création de services d'interrogation de données sur le web. Contrairement aux modèles de données classiques (notamment le modèle relationnel), les modèles de données semi-structurées assouplissent la séparation entre schéma et données en proposant une représentation unique. Le standard XML n'oblige pas la spécification d'un schéma ou type d'un document, mais cela s'avère, en général, très utile pour l'interrogation et l'échange de documents.

La mise à jour des documents XML nous permet de replacer plusieurs questions concernant la maintenance de la cohérence de la base vis à vis des contraintes. En effet, les contextes d'utilisation de XML sont habituellement très dynamiques et le type (ou schéma) peut être vu d'une manière plus flexible que celle adoptée dans le modèle relationnel. Le schéma XML est fréquemment sujet à des transformations pour s'adapter à de nouvelles demandes et de nouvelles fonctionnalités.

C'est donc dans ce nouveau cadre que je me suis penchée (à nouveau) sur les questions des mises à jour. Nos travaux ont commencé par la définition des primitives de mises à jour et la proposition d'une méthode pour la validation incrémentale par rapport aux contraintes de type. Étant donnée une liste de mises à jour sur un document initialement valide par rapport à un type, notre algorithme indique si les mises à jour violent cette validité. Dans nos travaux, nous avons considéré trois différentes façons de traiter ce problème :

1. En suivant une solution traditionnelle, l'algorithme de validation incrémentale considère les listes de mises à jour et rejette celles qui violent la validité.
2. Quand l'algorithme de validation incrémentale échoue, c'est-à-dire, quand il détermine qu'une mise à jour demandée viole le type, une routine de correction est activée pour adapter le document XML au type (par exemple en insérant ou en supprimant des éléments autres que ceux demandés par la mise à jour) tout en prenant en compte la mise à jour. Autrement dit, pour accepter une mise à jour en préservant la validité du document par rapport au type, cette approche propose des changements sur les données stockées[2]. La mise à jour a donc une priorité par rapport aux données déjà stockées.
3. Quand l'algorithme de validation incrémentale échoue, une routine propose une adaptation du type du document, de façon à accepter le document mis à jour en préservant la validité des autres documents originalement valides et non soumis à la mise à jour. Dans ce cas, un échec de l'algorithme de validation incrémentale est vu comme une demande d'un utilisateur spécialiste dans son domaine d'application qui souhaite changer les contraintes de schéma. La mise à jour est prioritaire et les contraintes peuvent être modifiées.

En plus des contraintes de type nous avons aussi considéré la validation incrémentale par rapport aux contraintes d'intégrité (clés, dépendances fonctionnelles, entre autres). Dans ce cas, pour l'instant, nous restons dans une approche classique où les mises à jour capables de violer la validité sont refusées.

Les mises à jour ont été, d'une certaine manière, le moteur de mon intérêt pour les services web. En effet, les services web sont des logiciels autonomes, accessibles via Internet, et sont typiquement conçus pour participer à des applications composées où l'interaction avec d'autres services est nécessaire. L'idée générale est donc de construire une plate-forme d'aide à la spécification, à l'implémentation et à la manipulation de services. À partir d'une bibliothèque de services, l'utilisateur peut spécifier un service composé, faire des vérifications sur sa composition et ensuite, éventuellement, faire des changements, par exemple, en remplaçant un service de la composition par un autre. Ainsi, l'idée de pouvoir spécifier et modifier des compositions de services nous a amené à la définition du langage PEWS (*Path Expression for Web Services*), ayant une sémantique formelle bien définie et permettant la spécification du comportement des interfaces des services web simples ou composés. La plate-forme mentionnée ci-dessus est un but à long terme, mais nous avons déjà implémenté des algorithmes permettant la vérification de la correction d'une composition.

Ainsi, la mise à jour et la maintenance de la cohérence d'une base de données par rapport à différents types de contraintes est encore un sujet d'actualité qui se présente de différentes manières, dans des

[2]Il est important de remarquer que , ici, le terme *données* fait référence aux nœuds de l'arbre XML, c'est-à-dire, les données définissant la structure du document. Il ne s'agit donc pas des valeurs associées aux éléments ou attributs XML, mais des éléments et des attributs eux mêmes.

contextes très variés. Mes travaux sont de modestes collaborations dans ce cadre. Plusieurs perspectives restent ouvertes notamment lorsqu'on considère des contraintes plus complexes. Considérer les particularités de chaque contexte en apportant l'expérience d'autres scénarios est un défi et une motivation qui, je crois, sont à la base du développement scientifique.

1.1 Les contributions

À mon arrivée à l'équipe BdTln (Base de données et Traitement des langues naturels) du LI, Campus Blois de l'Université François Rabelais de Tours, j'ai travaillé sur les entrepôts de données (ces travaux sont résumés dans ce chapitre mais ne seront pas traités dans ce mémoire), pour ensuite entamer des recherches dans le domaine de XML et services web. Un résumé des résultats obtenus dans chacun de ces travaux est proposé dans la suite.

1.1.1 Les aspects dynamiques des documents XML

Nous considérons le problème de la validation incrémentale par rapport aux contraintes structurelles (appelées type ou contraintes de schéma) et aux contraintes d'intégrité (concernant la sémantique des données). Les contraintes de schéma sont exprimées par une grammaire d'arbre alors que les contraintes d'intégrité sont exprimées par une grammaire d'attribut qui ajoute des règles sémantiques aux grammaires de schéma.

Validation incrémentale des documents XML par rapport aux contraintes de schéma et aux contraintes d'intégrité

Notre méthode concerne la validation incrémentale d'un document soumis à une suite de mises à jour. Cette validation est fondée sur l'exécution d'un automate d'arbres et consiste à vérifier s'il respecte les contraintes établies. La mise à jour d'un document XML présuppose la non violation des contraintes. Au lieu de faire, à chaque mise à jour envisagée, une validation intégrale (*from scratch*), nous proposons une méthode de *validation incrémentale* (c'est-à-dire, les tests sont effectués seulement sur la partie du document affectée par la mise à jour) capable de traiter des mises à jour multiples (et non une seule à la fois). La question de la validation incrémentale a été abordée principalement dans le cadre de deux doctorats sous mon encadrement scientifique à 45%.

- La thèse d'Ahmed Cheriat considère la validation incrémentale par rapport au type. Cette méthode parcourt l'arbre XML en même temps qu'une liste de mises à jour L. À chaque position p concernée par une mise à jour $u \in L$, l'algorithme simule l'exécution de u et continue son parcours. Des tests de validation sont déclenchés seulement sur les ancêtres de p.
- La thèse de Maria Adriana Vidigal de Lima (ex Abrão) concerne la validation incrémentale par rapport aux contraintes d'intégrité. Cet algorithme de validation suit le même raisonnement général du validateur de contraintes de schéma, mais déclenche des procédures de tests plus compliquées. Cette validation utilise une grammaire d'attribut pour raisonner et comparer les valeurs associées aux feuilles des arbres XML. Ces travaux introduisent aussi un formalisme homogène pour exprimer les différents types de contraintes d'intégrité. À partir de ce formalisme il nous est possible, en utilisant les grammaires d'attribut, de vérifier différents types de contraintes.

La plupart des outils de manipulation des documents XML ne possèdent pas de routine pour la validation incrémentale. Nos algorithmes ont été testés par rapport à des outils de validation intégrale (*from scratch*) et sont généralement plus efficaces.

Dans [BDHL03] nous avons introduit une méthode de validation *from scratch* permettant de traiter les attributs et les éléments d'un document XML. Ensuite, nous avons commencé à traiter le cas de la validation incrémentale par rapport aux schémas type DTD ([BH03]). Dans [BHM03] nous avons étendu nos automates d'arbre pour pouvoir valider un document XML par rapport aux clés. La validation incrémentale par rapport aux clés et clés étrangères est présentée dans [ABH+04]. Dans ces premiers travaux, seulement les mises à jour simples étaient considérées. Une version complète de notre méthode de vali-

dation incrémentale par rapport aux contraintes d'intégrité et aux contraintes de schéma (en considérant différents types de langage de schéma), sous des mises à jour multiples est publiée dans [BCH+07]. Un article de synthèse présentant différentes contraintes d'intégrité en XML via une syntaxe homogène est en cours d'évaluation [BHL07].

Correction des documents XML intégrée à la validation incrémentale

Ces travaux introduisent une méthode qui, au lieu de refuser les mises à jour qui invalident un document initialement valide (par rapport au type) propose des corrections sur ce document. La méthode de correction a été développée en deux étapes.

Dans un premier temps nous nous sommes restreints à un document très simple composé d'une racine et de ses fils. La correction du document correspond ainsi à la correction d'un mot (composé par la concaténation des étiquettes associées aux fils de la racine) par rapport a une grammaire (l'expression régulière représentant la contrainte de schéma associée à la racine). Soit A un mot valide sur lequel des mises à jour sont envisagées. Soit B le mot invalide résultant de la mise à jour de A. Étant donné un seuil d'erreur, la correction de B est réalisée par la recherche, dans un automate d'états finis, des plus proches voisins de A et de B (simultanément). En dépit d'une complexité exponentielle dans le pire des cas, nous avons obtenu de bons résultats expérimentaux calculés sur un large échantillon de mots avec des paramètres variables (la forme des expressions régulières, le seuil d'erreur, la taille du mot initialement valide et le nombre de mises a jour).

L'idée de la correction sur un mot a été étendue pour corriger l'arbre (le document XML) par rapport à un langage d'arbre (défini par le type du document XML). Cet algorithme introduit une méthode de correction d'un arbre par rapport à une grammaire d' arbre. En fait, étant donnés un langage d'arbre L et un arbre invalide $t \notin L$, notre algorithme calcule la distance d'édition entre t et $t' \in L$, dans un seuil donné.

Ce travail a été mené principalement dans le cadre de la thèse d'Ahmed Cheriat dont j'ai assuré l'encadrement scientifique à 45%. La thèse d'Ahmed Cheriat a été soutenue en novembre 2006. Ce travail a fait objet de deux publications dans des conférences internationales ([CSBH05] et [BCHS06b]) et un article dans une conférence nationale ([BCHS06a]).

Aide à l'évolution des schémas XML

Nous proposons un outil d'aide à l'évolution des schémas XML pour des utilisateurs non informaticiens mais spécialistes dans un domaine d'application. Un schéma XML est décrit par une grammaire régulière d'arbre où les expressions régulières définissent le langage (régulier) des fils d'un nœud (élément) dans un arbre XML. L'évolution du schéma XML correspond ainsi à l'évolution des expressions régulières E. Notre approche a deux caractéristiques principales :

1. L'évolution de schéma est déclenchée par une mise à jour prioritaire sur le document.
2. L'évolution de schéma préserve la cohérence des documents qui étaient valides par rapport au schéma original.

Deux versions de cette approche ont été développées. Une première version où les expressions régulières sont obtenues par un algorithme (nommé `GREC`) qui change l'automate d'état fini M_E associé à chaque expression E. Une deuxième version que nous appelons `dGREC`, obtient les mêmes solutions que `GREC` en travaillant seulement sur les expressions régulières, sans passer par la transformation des expressions régulières en automates.

Un programme datalog est une autre façon de représenter le schéma XML. Nous proposons une fonction de traduction qui engendre un programme datalog monadique à partir de la spécification d'un schéma XML. Ensuite, l'évolution de ce validateur datalog est assuré par des algorithmes similaires à `dGREC`. La version datalog offre une vision et une maintenance plus globale du schéma, en opposition à l'évolution des expressions régulières, une à une, des autres versions.

Ce travail a été principalement mené dans le cadre d'une thèse (D. Duarte) dont j'ai assuré l'encadrement scientifique (40%). La soutenance de cette thèse a eu lieu en juillet 2005. Ces recherches ont fait

l'objet de deux articles publiés dans des conférences internationales [BDH+04b, BDH+04a]. La version `dGREC` ainsi que sa version datalog sont présentées dans [dHM07]. Une version étendue de `GREC` qui traite le cas de plusieurs mises à jour a été soumise à un journal international [BDHM07].

1.1.2 Services web composés : spécifications et vérifications

Nous proposons le langage PEWS (*Path Expressions for Web Services*) de description d'interface des services web simples et composés. Le langage PEWS peut être vu comme un complément de WSDL (*Web Service Description Langange*) [CCMW01]. Les services web simples sont des expressions sur des opérations WSDL alors que les services web composés sont des expressions sur des services définis par PEWS.

Pour pouvoir tester statiquement des propriétés liées à la composition des services, nous proposons l'utilisation de la théorie des traces, introduite dans [Maz87, Maz95]. Les traces montrent de façon séquentielle le comportement non séquentiel d'un système concurrent. Néanmoins, même si les traces nous permettent d'exprimer de façon élégante la communication entre services, l'implémentation des outils de vérification en utilisant cette théorie n'est pas très efficace. Les graphes de dépendances ([Maz95]), équivalents aux traces, ont été utilisés pour surmonter cette difficulté.

Cette recherche est menée dans le cadre de la thèse de Cheikh Ba, dont j'assure l'encadrement scientifique (95%), en collaboration avec le BRGM (Bureau de Recherches Géologiques et Minières) et avec le groupe de travail de Martin Musicante (DIMAp, UFRN, Brésil). La spécification du langage PEWS pour les services simples à été publiée dans [BHM05] (la version étendue de ce papier est dans [BCHM05]). L'utilisation de la théorie de traces pour les vérifications sur les compositions est proposée dans [BHM06]. Dans [BH08] nous proposons l'utilisation des graphes de dépendances pour implémenter nos vérifications.

1.1.3 Maintenance des entrepôts de données temporelles

Dans ces travaux, nous proposons une méthode pour la maintenance incrémentale des entrepôts de données contenant des vues temporelles sur des sources non temporelles. Pour éviter le stockage de tout l'historique des bases sources, notre approche utilise des relations auxiliaires, stockées dans l'entrepôt. Une importante contribution de nos recherches est l'introduction d'un formalisme pour la spécification des entrepôts de données temporelles contenant toutes les informations nécessaires à sa maintenance. Selon ce formalisme, un entrepôt de données temporelles **W** est une paire de deux ensembles de vues : le *composant matérialisé* et *composant virtuel*. Le composant matérialisé de **W** représente l'ensemble des vues physiquement stockées dans l'entrepôt. Le composant virtuel de **W** est un ensemble d'expressions non-temporelles comportant seulement des relations stockées dans le composant matérialisé. Plusieurs dispositifs de notre approche la rendent particulièrement attrayante comme méthode de maintenance des entrepôt : (*i*) Nous n'avons pas besoin de stocker l'historique entier des bases de données de sources. (*ii*) La maintenance de l'entrepôt temporel est réduite à maintenir le composant matérialisé (non-temporel). (*iii*) La maintenance du composant matérialisé est faite à partir des informations qu'il stocke.

Notre algorithme combine deux techniques indépendantes proposées dans [LLSV01] et [Cho92] et fondées sur les relations auxiliaires. Ces relations stockent seulement la partie de l'historique nécessaire pour la maintenance des vues de l'entrepôt. La maintenance de l'entrepôt est effectuée sans avoir besoin de consulter les sources.

Ce travail a été réalisé dans le cadre d'une collaboration internationale. Les premiers résultats ont été publiés dans [AH00] alors qu'une version étendue, considérant des vues spécifiées par des opérateurs d'agrégation, est parue dans [AH04].

1.2 Bilan

Cette section présente un historique de mon parcours scientifique puis un résumé de mes encadrements de thèse et la liste des publications concernant mes travaux de recherche depuis mon arrivée à l'Université

1.2.1 Historique des activités scientifiques et collaborations

Après mon doctorat, de retour au Brésil auprès de l'*Universidade Federal do Paraná* à laquelle j'étais déjà attachée avant le début de ma thèse, j'ai contribué à la mise en place d' un groupe de recherche sur les entrepôts de données au centre du département informatique. À cette époque j'ai aussi commencé une collaboration avec Sandra de Amo, enseignant chercheur à l'*Universidade Federal de de Uberlândia*, Brésil, sur la maintenance des entrepôts de données temporelles.

À mon arrivée à l'équipe BdTln du LI de l'Université François Rabelais de Tours (en 1998), un groupe dans le domaine du *data mining* était déjà formé et les travaux étaient bien avancés. Dans un premier temps, j'ai continué à travailler sur les mises à jour des entrepôts de données.

Dans la recherche de thèmes qui pouvaient fédérer différentes capacités de notre équipe nous avons entamé, Béatrice Bouchou et moi, une nouvelle collaboration dans le domaine du XML. L'acceptation du projet Capes-Cofecub (que j'avais élaboré auparavant avec la collaboration de Dominique Laurent) a donné une dynamique à nos recherches puisque nous avons eu deux thésards du projet intéressés par nos thèmes. Avec l'appui de Dominique Laurent nous avons donc pris la responsabilité de l'encadrement de Denio Duarte (thèse soutenue en 2005) et Maria Adriana Vidigal de Lima (soutenance prévue pour 2007).

Ainsi, en 2001, un nouvel axe de recherche sur XML est né dans l'équipe BdTln. Cet axe a pu se développer engendrant des collaborations internationales en particulier celle de Martin Musicante, aujourd'hui enseignant chercheur à l'*Universidade Federal do Rio Grande do Norte* (UFRN), Brésil et membre du projet Capes-Cofecub. Un nouveau thésard, Ahmed Cheriat a intégré le groupe (co-encadré par Béatrice Bouchou et moi, ayant comme directeur de recherche Dominique Laurent). Le travail avec Ahmed (portant sur la correction des documents XML) a attiré l'attention d'une autre collègue Tln, Agata Savary. Ses connaissances dans le domaine de la correction des chaînes de caractères ont été essentielles pour le développement de la thèse d'Ahmed Cheriat (thèse soutenue en novembre 2006).

Les activités du groupe se sont élargies à partir de 2004, période à laquelle j'ai commencé à m'investir aussi sur les problèmes liés aux services web. Cette nouvelle problématique a attiré l'attention d'un de mes étudiants du Master, qui a souhaité entamé une thèse dans le domaine, ainsi que de Martin Musicante qui commençait des recherches dans ce domaine, au Brésil. Suite à un contact avec le BRGM (Bureau de Recherches Géologiques et Minières), aussi intéressé par certains aspects des services web, j'ai pris la responsabilité intégrale de l'encadrement de la thèse de Cheikh Ba (financée par une bourse BRGM -Région Centre, ayant comme directeur de thèse Jean-Yves Antoine).

Aujourd'hui, l'axe de recherche données XML et services web est une thématique importante dans l'équipe BdTln, témoignant aussi de la possibilité de collaboration entre les domaines des bases de données et du traitement des langues naturels.

1.2.2 Encadrement de thèses

- *Cheick Ba* : Doctorat en cours depuis décembre 2004 sur le thème de la composition et manipulation des services web. Nous comptons avec une bourse région co-financée par le BRGM depuis octobre 2005. Soutenance de thèse prévue pour 2008.
  Directeur de thèse : Jean-Yves Antoine (LI - Université François Rabelais).
  Encadrement : Mírian Halfeld Ferrari : 95%, Jean-Yves Antoine : 5%

- *Ahmed Cheriat* : **Thèse soutenue le 30 novembre 2006.** Ce doctorat a été financé par une bourse région et un 1/2 poste d'ATER à l'UFR Sciences, Campus Blois.
  Titre de la thèse : Une méthode de correction de la structure de documents XML dans le cadre d'une validation incrémentale
  Directeur de thèse : Dominique Laurent (Université de Cergy Pontoise).
  Encadrement : Mírian Halfeld Ferrari : 45%, Béatrice Bouchou : 45% et Dominique Laurent 10%.

- *Denio Duarte* : **Thèse soutenue le 4 juillet 2005.** Ce doctorat a été financé par une bourse

Capes-Cofecub (4 ans).
Titre de la thèse : Une méthode pour l'évolution de schémas XML préservant la validité des documents.
Directeur de thèse : Dominique Laurent (Université de Cergy Pontoise).
Encadrement Mírian Halfeld Ferrari : 40%, Béatrice Bouchou : 40% et Dominique Laurent 20%

– *Maria Adriana Vidigal de Lima* : Doctorat en cours depuis décembre 2001 (avec un congé maternité) sur le thème de la validation incrémentale des contraintes d'intégrité sur les documents XML. Ce doctorat a été partiellement financé par une bourse Capes-Cofecub (2 ans). Depuis septembre 2005, Maria Adriana est de retour au Brésil où elle a un poste d'enseignant-chercheur à l'Université Catholique de Minas Gerais. Suite à des problèmes personnels, elle a pris du retard dans sa thèse. Sa soutenance est prévue pour 2007.
Directeur de thèse : Dominique Laurent (Université de Cergy Pontoise).
Encadrement : Mírian Halfeld Ferrari : 45%, Béatrice Bouchou : 45% et Dominique Laurent 10%

1.2.3 Publications

Remarques :
(1) Dans mes travaux, l'ordre alphabétique de noms est le *standard* pour la présentation des auteurs d'un article. Les articles qui ne suivent pas le standard sont signalés par une $\star$.
(2) Publications depuis mon arrivée à l'Université François Rabelais de Tours. Période de 1998 à 2007.

Revues internationales avec comité de lecture

1. [$BCHLLM07$] Béatrice Bouchou, Ahmed Cheriat, Mírian Halfeld Ferrari, Dominique Laurent, Maria-Adriana Lima and M. Musicante, *Efficient Constraint Validation for Updated XML Databases*, Informatica, volume 31, number 3, pages 285 – 310, 2007.
2. [$dLHM07$] Robson da Luz and Mírian Halfeld Ferrari and Martin A. Musicante, *Regular Expression Transformations to Extend Regular Languages (With Application to a Datalog XML Schema Validator)*, Journal of Algorithms (Special Issue), Volume 62, Issues 3 – 4, (July-October 2007), pages 148 – 167, Elsevier, 2007.
3. [$BCHM05$] Cheikh Ba ; Marcos Aurélio Carrero ; Mírian Halfeld Ferrari and Martin Musicante ; *PEWS : A New Language for Building Web Service Interfaces*, Journal of Universal Computer Science, Volume 11, No. 7, Special Issue, pages 1215 – 1233, July, 2005.
4. [$AH04$] Sandra de Amo and Mírian Halfeld Ferrari Alves *Incremental Maintenance of Data Warehouses Based on Past Temporal Logic Operators*, Journal of Universal Computer Science, vol 10, no. 9, pages 1035 – 1064, 2004.
5. [$HLS98$] Mírian Halfeld Ferrari Alves ; Dominique Laurent and Nicolas Spyratos. *Update Rules in Datalog Programs*, *Journal of Logic and Computation*, Vol 8, numéro 6, pages 745 – 775, december 1998.

Actes des colloques internationaux avec comité de programme

1. [$BaH08$] Cheikh Ba and Mirian Halfeld Ferrari, *Dependence Graphs for Verifications of Web Service Compositions with PEWS*, accepted in the ACM Symposium on Applied Computing (SAC - SIGAPP), special track on Web Technologies, to appear, 2008.
2. [$BHM06$] Cheikh Ba, Mirian Halfeld Ferrari and Martin Musicante *Composing Web Services with PEWS : a trace-theoretical approach* ; The 4th IEEE European Conference on Web Services (ECOWS) ; December 2006.
3. [$BCHS06$] Béatrice Bouchou, Ahmed Cheriat, Mirian Halfeld Ferrari and Agata Savary *XML Document Correction : Incremental Approach Activated by Schema Validation* ; Proceedings of the International Database Engineering and Applications Symposium - IDEAS 2006.
4. [$CSBH05$] A. Cheriat, A. Savary, B. Bouchou and M. Halfeld Ferrari ; *Incremental String Correction : Towards Correction of XML Documents*, in Proceedings of the Prague Stringology Conference '05 (PSC'05), Prague, August 2005 $\star$.

5. [$ABHLM04$] Maria Adriana Abrão, Béatrice Bouchou, Mírian Halfeld Ferrari Alves, Dominique Laurent and Martin Musicante ; *Incremental Constraint Checking for XML Documents*, Database and XML Technologies : Second International XML Database Symposium (XSym), Toronto, Canada, August 29-30, LNCS (Lecture Notes in Computer Science) Volume 3186, 2004.
6. [$BDHLM04a$] Béatrice Bouchou, Denio Duarte, Mírian Halfeld Ferrari Alves, Dominique Laurent and Martin Musicante ; *Conservative Extensions of Regular Languages*, XXIV International Conference of the Chilean Computer Science Society (SCCC 2004).
7. [$BDHLM04b$] Béatrice Bouchou, Denio Duarte, Mírian Halfeld Ferrari Alves, Dominique Laurent and Martin Musicante ; *Schema Evolution for XML : A Consistency-preserving Approach*, 29th International Symposium, Mathematical Foundations of Computer Science (MFCS), Prague, Czech Republic, August 22-27, LNCS (Lecture Notes in Computer Science) Volume 3153, 2004.
8. [$BFHV04$] Mateus Barcellos da Costa, Rodolfo Ferreira Resende, Mírian Halfeld Ferrari and Marcelo Vieira Segatto ; *Business to Business Transaction Modeling and WWW Support*, 2nd International Conference on Business Process Management - BPM, 2004 ; LNCS Volume 3080/2004. ⋆
9. [$BCHJL04b$] Béatrice Bouchou, Ahmed Cheriat, Mírian Halfeld Ferrari Alves, Tao Jen and Dominique Laurent ; *XRM : An XML-based Language for Rule Mining Systems*, International Conference on Enterprise Information System - ICEIS, 2004.
10. [$BH03$] Béatrice Bouchou and Mírian Halfeld Ferrari Alves, *Updates and incremental validation of XML documents*, The 9th International Workshop on Data Base Programming Languages (DBPL), LNCS (Lecture Notes in Computer Science) Volume 2921, 2003.
11. [$BHM03$] Béatrice Bouchou, Mírian Halfeld Ferrari Alves and Martin A. Musicante, *Tree Automata to Verify Key Constraints*, Web and Databases (WebDB), San Diego, CA, USA, 2003. http ://www.cse.ogi.edu/webdb03/index.htm
12. [$BDHL03$] Béatrice Bouchou and Denio Duarte and Mírian Halfeld Ferrari Alves and Dominique Laurent, *Extending Tree Automata to Model XML validation under Element and Attribute Constraints*, 5th International Conference On Enterprise Information Systems (ICEIS), 2003.
13. [$AH00$] Sandra de Amo and Mírian Halfeld Ferrari Alves ; *Efficient Maintenance of Temporal Data Warehouses*, Proceedings of the International Database Engineering and Applications Symposium - IDEAS 2000.

Actes des colloques nationaux avec comité de programme

1. [$BCHS06b$] Béatrice Bouchou, Ahmed Cheriat, Mirian Halfeld Ferrari and Agata Savary, *Integrating Correction into Incremental Validation*, Journées de Bases de Données Avancées, BDA 2006.
2. [$BHM05$] Cheikh Ba, Mírian Halfeld Ferrari and Martin Musicante ; *Building Web Service Interfaces using Predicate Path Expressions* ; SBLP 2005 - 9th Brazilian Symposium on Programming, 2005.
3. [$BFSH$] Mateus Barcellos Costa ; Rodolfo Ferreira Resende, Pedro dos Santos Neto and Mírian Halfeld Ferrari ; *Utilização de aspectos no desenvolvimento de aplicações baseadas em serviços web (L'utilisation des aspects dans le développement des services web)*, Actes du premier *Workshop Brasileiro de Desenvolvimento Orientado a Aspectos*, Brasília, 2004 (en Portugais). ⋆

1.3 Structure du mémoire

La suite de ce mémoire est organisée en sept chapitres. Les deux premiers chapitres sont consacrés aux préliminaires. Le chapitre 2 introduit certaines définitions et notations que nous utilisons dans le contexte de XML et propose un panorama succinct des formalismes pour les arbres d'arité variable. Le chapitre 3 présente la sémantique de nos opérations de mises à jour en XML.

La validation incrémentale et nos différentes manières de maintenir la cohérence des documents XML sont décrites dans les chapitres 4, 5 et 6. Le chapitre 4 décrit notre méthode de validation incrémentale par rapport aux contraintes de schéma, ainsi que notre méthode de correction des documents XML. Le chapitre 5 traite l'évolution des schémas XML, déclenchée par des mises à jour sur les documents. Un parallèle entre les deux versions de cette approche est proposé. Le chapitre 6 est consacré aux contraintes

d'intégrité dans un document XML et présente une synthèse de nos travaux, tout en proposant un nouveau regard sur le langage de définition des contraintes. Le chapitre 7 est complètement indépendant des chapitres précédents. Il résume nos premiers résultats dans le domaine des services web. Finalement, le dernier chapitre est consacré aux conclusions et perspectives.

Chapitre 2

XML et les formalismes pour les arbres d'arité variable

XML (Extensible Markup Language [XMLa]) est une notation pour représenter des arbres dans un format textuel ([KSS03]). Dans cette représentation, le texte correspondant à un nœud est délimité par des balises de début et de fin. Le texte entre deux balises décrit les fils d'un nœud. À la fin du chapitre, la figure 2.2 montre un exemple d'un document dans le format texte et dans le format arborescent.

XML est devenu le standard pour l'échange de données, *the lingua franca of the Web*. Cela peut étonner puisqu'il s'agit de remettre en place une idée ancienne (les arbres représentés par une syntaxe linéaire) [KSS03]. Mais, ces vieux concepts ont été enrichis par de nouvelles propositions, notamment par le rapprochement entre les fondements des bases de données et les formalismes pour la manipulation des arbres.

Les données XML sont typiquement associées aux arbres *d'arité non bornée ou variable*. Dans ces arbres, contrairement aux arbres *d'arité borné ou fixe* ([GS97, CDG$^+$02, Tho90, Tho97]) où le nombre de fils d'un nœud est déterminé par le label du nœud, le nombre (fini) d'enfants qu'un nœud peut avoir est variable. Même si les premiers travaux sur les arbres d'arité variable datent des années 70 [Tak75, Tha67] l'intérêt et l'étude systématique de ces structures sont relativement récents [BW98b, KSS03, MLM01, Nev02, Via01] et motivés surtout par XML.

Ce chapitre introduit certaines définitions et notations utilisées dans ce mémoire tout en proposant un panorama succinct sur les liaisons entre les arbres ordonnés d'arité variable, la logique du premier ordre (FO) et la logique monadique du second ordre (MSO), les programmes datalog et les automates d'arbres. Ce chapitre est un résumé, inspiré des aperçus plus détaillés proposés dans [KSS03, Lib06, MLMK05].

2.1 Arbres d'arité variable, logiques et datalog

Un arbre est composé d'un *domaine d'arbre* avec une *fonction d'étiquetage* sur les nœuds.

Définition 2.1.1 - Arbres d'arité variable ou non bornée (*Unranked trees*) : Un *domaine d'arbre* D est un sous ensemble de $\mathbb{N}^*$ qui respecte les propriétés suivantes : (1) D est fermé sur les préfixes, c'est-à-dire, si $u, u' \in \mathbb{N}^*$ et u est préfixe de u' (noté $u \preceq u'$) et $u' \in D$, alors $u \in D$ et (2) Si $j \geq 0$ et $u.j \in D$ et $0 \leq i \leq j$ (où $i, j \in \mathbb{N}$) alors $u.i \in D$. Chaque élément de D est appelé *position*.

Soit Σ un alphabet. Un *arbre d'arité non bornée* (*unranked tree*) est une paire $T = (D, t)$ où D est un domaine d'arbre et t est une fonction d'étiquetage $t : D \to \Sigma \cup \{\lambda\}$ (λ est un symbole spécial). La racine d'un arbre est à la position ϵ. L'arbre vide est définie par $T = (\{\epsilon\}, \{(\epsilon, \lambda)\})$ □

La figure 2.1(a), illustre un arbre d'arité fixe où tous les nœuds a ont 2 enfants et tous les nœuds b ont 1 enfant. Dans la figure 2.1(b) un arbre d'arité variable est illustré. Chaque noeud est représenté par une *position* associée à un label : la racine a le label a, c'est-à-dire, $t(\epsilon) = a$, et $t(0) = a$, $t(1) = b$, $t(2) = b$, $t(2.0) = b$ et ainsi de suite. Remarquer que la fonction d'étiquetage pour l'arbre T de la figure 2.1(b) peut aussi être représentée par : $t = \{(\epsilon, a), (0, a), (1, b), (2, b), (0.0, d), (0.1, d), (1.0, d), (2.0, d), (2.1, d),$

$(2.0.0, x)\}$.

Selon la définition 2.1.1 les positions dans les arbres d'arité variable sont des éléments de $\mathbb{N}^*$, c'est-à-dire, des chaînes de caractères finies dont les caractères sont les nombres naturels. Ainsi, une chaîne $u = i_0.i_1 \dots i_n$ définit un chemin qui commence à la racine $i_0 = \epsilon$ passe par le $i_1$ème fils de i_0 et ainsi de suite jusqu'à arriver au $i_n$ème fils de $v = i_0.i_1 \dots i_{n-1}$. Dans la figure 2.1(b), la position 2.0.0 représente un chemin pour arriver au noeud étiqueté x, à partir de la racine.

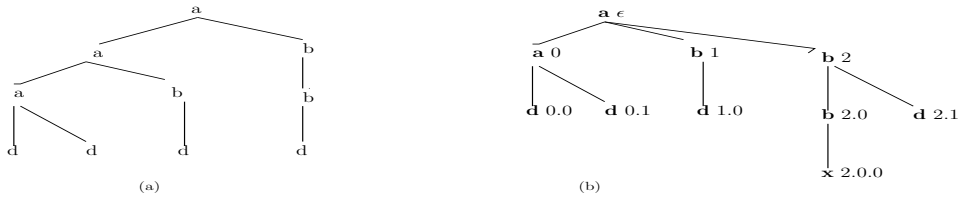


FIG. 2.1: (a) Arbre d'arité fixe (b) Arbre d'arité variable

Dans ce mémoire, par souci de simplification de certaines explications (en évitant de dessiner des arbres d'arité non bornée) nous allons utiliser (quelques fois) la représentation d'un arbre via une chaîne de caractères avec des parenthèses imbriquées, c'est-à-dire, une représentation similaire à une chaîne de Dick [BW04]. Par exemple, la figure 2.1(b) peut être représentée par $a(a(d, d), b(d), b(b(x), d))$.

Il est important de remarquer que, dans le contexte XML, la définition d'arbre d'arité variable impose, en général, un ordre entre frères et, dans ce cas, nous parlons *d'arbre ordonné*. Quand cet ordre n'est pas important, nous avons un *arbre non ordonné*. Dans ce mémoire, sauf précision contraire, nous considérons les arbres d'arité variable ordonnés.

Nous introduisons aussi la notion de sous-arbre d'un arbre donné.

Définition 2.1.2 - Sous-arbre d'un arbre d'arité variable (*subtree*) : Étant donné un arbre $T = (D, t)$ nous dénotons $T_p^{sub} = (D_p^{sub}, t_p^{sub})$ le sous-arbre dont la racine est à la position $p \in D$, c'est-à-dire, le sous ensemble $D_p^{sub} \subseteq D$ contient toutes le positions v telles que $p \preceq v$ et $t_p^{sub} = \{(v, t(v)) \mid v \in D_p^{sub}\}$. □

Remarquer qu'un sous-arbre n'est pas un arbre car sa racine n'est pas dans la position ϵ. Par exemple, dans la figure 2.1(b) $t_2^{sub} = \{\ (2, b),\ (2.0, b),\ (2.1, d),\ (2.0.0, x)\}$.

Dans le contexte XML, il semble naturel d'utiliser les formalismes logiques pour exprimer des requêtes de façon déclarative, c'est-à-dire, comme paradigmes des langages de requêtes et d'appliquer les formalismes procéduraux, comme les automates, dans l'évaluation des requêtes [Lib06].

En termes de formalisme logique, nous remarquons, par exemple, que différents fragments de XPath [XPab] sont liés à la logique du premier ordre (FO). XPath 1.0 permet la sélection d'un ensemble de noeuds (*answer set*) dans un arbre XML via la définition d'une expression de chemin qui utilise différents axes de "navigation" sur l'arbre XML. La question est de savoir si tous les chemins exprimables en XPath sont aussi exprimables en FO et vice versa. XPath 2.0 est plus expressif [MdR05] et manipule des listes [XPaa, XPac]. En effet, comme la FO n'est pas suffisante pour exprimer toutes les expressions régulières (par exemple $(aa)^*$), nous pouvons imaginer un langage de navigation (ou un langage de requêtes) sur les documents XML dont le pouvoir d'expression dépasse la FO car l'utilisation des quantificateurs du second ordre s'impose. La logique monadique du second ordre (MSO) peut être vue, dans la plupart des cas, comme la base pour les langages de requêtes et de spécification des schémas XML : une DTD ou une définition XSD (XML Schema) sont équivalentes à des formules en MSO ; plusieurs langages dont le but est l'extraction des données des documents XML ont le même pouvoir d'expression que les requêtes MSO unaires.

L'étude classique du pouvoir d'expression des différentes logiques sur les arbres comporte la représentation des arbres par des structures sur un vocabulaire fixe.

Définition 2.1.3 - Structure d'arbre ordonné : Un arbre *ordonné d'arité non bornée* $T = (D, t)$ peut être défini comme une structure (ou modèle) $T = \langle D, \prec_{ch}^*, \prec_{ns}^*, (P_a)_{a \in \Sigma \cup \{\lambda\}} \rangle$ où :

1. D est un domaine d'arbre ;
2. La relation *descendent* ($\prec_{ch}^*$) est la fermeture transitive de la relation *child* ($\prec_{ch}$).
   Pour $u, u' \in D$, nous définissons :

– $u \prec_{ch} u'$ ssi $u' = u.i$ avec $i \in \mathbb{N}$.
– $u \prec^*_{ch} u'$ ssi u est un préfixe[1] de u' ou $u = u'$.

3. La relation *linear order of siblings* ($\prec^*_{ns}$) est la fermeture transitive de la relation *next-sibling* ($\prec_{ns}$). Pour $u, u' \in D$, nous définissons :
   – $u \prec_{ns} u' \Leftrightarrow u = u_0.i$ et $u' = u_0.(i+1)$ pour un $u_0 \in \mathbb{N}^*$ et $i \in \mathbb{N}$.
   – $u \prec^*_{ns} u'$ ssi $u = u_0.i$ et $u' = u_0.j$ et $i \leq j$, avec $i, j \in \mathbb{N}$ et $u_0 \in \mathbb{N}^*$.
4. P_a sont des ensembles disjoints de positions dont l'union est le domaine D (définissons $P_a = \{u \in D \mid t(u) = a\}$). □

Étant donné un arbre T nous sommes, en général, intéressés à savoir si T est un modèle pour une requête φ, exprimée comme une formule logique. Dans ce contexte, la notion de *requête définissable dans une logique* est introduite.

Définition 2.1.4 - Requête Φ-définissable [Lib06] : Étant donnée une logique Φ, une requête booléenne (qui définit un langage d'arbres $\mathcal{L}$, c'est-à-dire, un ensemble d'arbres) est Φ-définissable s'il existe une formule φ de Φ telle que $T \in \mathcal{L}$ ssi $T \models \varphi$ (c'est-à-dire, φ est vrai dans T ou T est un modèle pour φ).
De façon similaire, une requête unaire $\mathcal{Q}$ est Φ-définissable s'il existe une formule $\phi(x)$ de Φ telle que, pour tout arbre T et un nœud s dans T, $s \in \mathcal{Q}(T)$ ssi $T \models \phi(s)$ (où $\mathcal{Q}(T)$ est le résultat de l'évaluation de la requête $\mathcal{Q}$ sur l'arbre T). □

Par exemple, supposons une requête comme une expression de chemin qui nous permet de sélectionner des noeuds d'un arbre T ayant les caractéristiques suivantes : les nœuds recherchés ont pour étiquette a ; ils ont un ancêtre c qui, à son tour, a un fils étiqueté b. Cette expression peut être exprimée par une formule du premier ordre, à savoir :

$$\varphi(x) \equiv P_a(x) \wedge \exists y \exists z (P_c(y) \wedge P_b(z) \wedge (y \prec^*_{ch} x) \wedge \\ (\neg(y = z) \wedge (y \prec^*_{ch} z) \wedge \forall v((y \prec^*_{ch} v) \wedge (v \prec^*_{ch} z) \rightarrow (y = v) \vee (z = v))$$

où x est une variable libre qui correspond aux réponses recherchées et la sous-expression $(\neg(y = z) \wedge (y \prec^*_{ch} z) \wedge \forall v((y \prec^*_{ch} v) \wedge (v \prec^*_{ch} z) \rightarrow (y = v) \vee (z = v))$ définit z comme fils[2] de y. Le *answer set* est défini par l'ensemble des positions x du domaine d'un arbre T tel que $T \models \varphi(x)$.

L'étude du pouvoir d'expression des langages de navigation ou de requêtes s'avère très utile pour la compréhension, la comparaison, la conception (ou l'enrichissement) de ces langages. Dans le cadre de cette étude, il est usuel de considérer la traduction des requêtes (exprimées dans un langage XML) en une logique Φ. Par exemple, XPath [XPab] est un langage qui permet la sélection des noeuds dans un document XML. XPath1 a un rôle important dans d'autres langages de requête XML, comme XSLT [XSL] et XQuery [XQu] ainsi que dans des langages de spécification de contraintes comme XML Schema [XMLb]. Par exemple, l'expression XPath

$$//c[/b]//a$$

fournit les noeuds étiquetés a qui sont descendants d'un noeud c qui, à son tours, a un fils b. Dans cette notation (simplifiée), // représente l'axe descendant, / représente l'axe fils et [.] est un *qualificateur*, c'est-à-dire, une condition qui doit être vérifiée par le noeud.

Les aspects de navigation de XPath peuvent être exprimés par la logique du premier ordre et ce fait a comme conséquence : d'un coté, l'attention portée à des logiques ayant un pouvoir d'expression égal ou inférieur à la logique du premier ordre (voir [Lib06] pour un aperçu des relations entre la FO sur les arbres et les logique temporelles) ; d'un autre coté la motivation par l'étude de la logique du premier ordre sur les arbres (une forme de continuation des résultats concernant FO sur les chaînes de caractères [Lib06, Tho97]). Rappelons nous que sur les chaînes - qui peuvent être vues comme des arbres unaires - la FO définit précisément les langages *star-free* ([Lib06, Tho97]).

Les travaux, comme [BFK03, BK07, Mar04, MdR05], concernant le pouvoir d'expression des fragments XPath et la logique du premier ordre sont assez récents. Par exemple, Core XPath (proposé dans [GKP03]) est étudié dans [BFK03] qui nous fournit l'étude des fragments positifs sans l'axe *sibling*, et dans [MdR05] qui présente une extension de [BFK03] avec l'axe *sibling* et la négation. Une hiérarchie des fragments

[1] REMARQUE : Dans ce mémoire, nous utilisons le symbole $\preceq$ de façon équivalente à $\prec^*_{ch}$. Le choix est fait selon le contexte, par soucis de clarté. Remarquer aussi que $\prec$ indique un préfixe strict, c'est-à-dire, si $p \prec p'$ alors $p \preceq p' \wedge p \neq p'$.
[2] Remarquer que dans la définitions 2.1.3, la structure T possède la relation $\prec^*_{ch}$ et non $\prec_{ch}$.

XPath a été établie. Core XPath est équivalent à la logique du premier ordre restreinte à deux variables[3], avec les axes *child*, *descendant* et *next-sibling* [MdR05]. Conditional XPath ([Mar04]) correspond à la FO et Regular XPath a un pouvoir d'expression supérieur à la FO (entre FO et MSO).

La logique monadique du second ordre (MSO), qui permet la quantification sur les ensembles, rend possible l'expression de requêtes plus puissantes. Soit la formule ([Lib06]) ci-dessous[4] où les ensembles sont représentés par des majuscules (X, Y) ; $X(x)$ est une formule atomique avec une variable de second ordre X et une variable du premier ordre x et les quantificateurs utilisés sont du premier (par exemple, $\exists x\varphi$) ou du second ordre (par exemple, $\exists X\varphi$) [Lib04].

$\varphi_{odd}(x,y) = x \prec^*_{ch} y \wedge \varphi_0(x,y)$ où

$$\varphi_0(x,y) = \exists X \exists Y \left(\begin{array}{l} \forall z((x \prec^*_{ch} z \prec^*_{ch} y) \rightarrow (X(z) \leftrightarrow \neg Y(z))) \\ \wedge (X(x) \wedge Y(y)) \\ \wedge \forall z \forall v (x \prec^*_{ch} z \prec_{ch} v \prec^*_{ch} y \rightarrow ((X(z) \rightarrow Y(v)) \wedge (Y(z) \rightarrow X(v)))) \end{array} \right)$$

Cette formule dit qu'il existe deux ensembles X et Y qui partagent le chemin entre le noeud x et son descendant y de la manière suivante : $x \in X$, $y \in Y$, le successeur de chaque élément de X est dans Y et le successeur de chaque élément de Y est dans X. En d'autres mots, la formule définit les chemins de longueur impaire de x à y. Cette requête aurait pu être écrite en partant du principe que les relations de base, établies dans la définition 2.1.3, étaient les $\prec_{ch}$ et $\prec_{ns}$ (sans la fermeture transitive).

Tous les résultats sur MSO peuvent, en effet, être présentés en utilisant, $\prec_{ch}$ et $\prec_{ns}$ comme les relations de base. Cela n'est pas vrai pour FO : les relations $\prec_{ch}$ et $\prec_{ns}$ sont définissables à partir des relations $\prec^*_{ch}$ et $\prec^*_{ns}$, mais, c'est connu, l'inverse n'est pas vrai ([Lib04]).

MSO est une logique de base pour les arbres d'arité variable ordonnés dû à sa connexion avec les langages réguliers. Les langages réguliers d'arbres sont essentiels dans le domaine XML. Considérons, par exemple, les schémas XML décrits via des langages de schéma (actuels) qui nous permettent d'établir des contraintes structurelles sur les documents XML. Ces schémas XML définissent des langages réguliers d'arbres.

Le théorème ci-dessous est déjà un classique et correspond à une extension des résultats obtenus avec les chaînes de caractères dans [Büc60] et avec les langages réguliers d'arbres binaire dans [Don70, TW68]. Voir [Lib04] pour un très bon aperçu dans ce domaine.

Théorème 2.1.1 [Nev99] : *Un langage d'arbres d'arité variable est régulier ssi il est MSO-définissable.* □

Comme les langages réguliers d'arbres sont ceux reconnus par les automates d'arbres (section 2.2), il existe aussi un lien important entre les automates, MSO et les langages de schémas. Donc, supposant que $\mathcal{L}(\Upsilon)$ est un langage d'arbres défini par un schéma Υ donné, $\mathcal{L}(\Upsilon)$ est MSO-définissable. Même si la complexité de MSO est élevée, pour certaines structures, comme les arbres, la liaison avec les automates fournit des limites intéressantes en termes de complexité. En effet, toute expression MSO φ peut être transformée en automate d'arbres déterministe $\mathcal{A}_\varphi$ qui reconnaît un arbre T ssi $T \models \varphi$. Ce résultat nous fournit un algorithme en $O(|T|)$ pour vérifier si $T \models \varphi$, si φ est fixe. Néanmoins, le problème est que la conversion d'une formule MSO en $\mathcal{A}_\varphi$ est non élémentaire, c'est-à-dire, la taille de $\mathcal{A}_\varphi$ (même pour les chaînes de caractères) ne peut pas être limitée par une fonction élémentaire dans $|\varphi|$ ([Lib04, Lib06])[5]. Ces résultats nous attirent vers l'utilisation d'une logique Φ ayant le même pouvoir d'expression que la MSO mais permettant une vérification de modèle plus efficace ([Lib06]) .

Il est prouvé dans [GK02] que datalog monadique est équivalent à MSO dans sa capacité à exprimer les requêtes unaires sur les arbres. Dans ce mémoire nous nous intéressons à l'usage de datalog dans la validation des contraintes de schéma XML.

[3] Un ensemble des formules $\varphi(x)$ de la logique du premier ordre contenant au plus deux variables, avec au plus une variable libre x.

[4] Remarquer que dans la formule nous utilisons $\prec_{ch}$ pour indiquer la relation père fils. Cette relation peut être traduite en $\prec^*_{ch}$ comme montré précedament.

[5] Ces résultats ont été publiés par [SM02] ; dans [FG02] les auteurs montrent qu'il n'existe aucun algorithme qui teste si $T \models \varphi$ en temps $O(f(|\varphi|) \times |T|)$, où f est une fonction élémentaire ; sauf si $P_{\text{TIME}} = NP$. Nous trouvons dans [Lib04, Lib06] un aperçu sur ces aspects.

Un programme datalog est comme un programme prolog sans les symboles de fonction. Il est composé d'une séquence des règles de la forme $H(u_0) \leftarrow R_1(u_1), \ldots, R_k(u_k)$ où $H(u_0)$ et $R_i(u_i)$ sont des atomes (littéraux positifs) avec des n-uplets (n-aires) libres $u_0, \ldots, u_k$ [AHV95, CGT90]. Le prédicat H est la *tête* (*head*) de la règle et la conjonction $R_1, \ldots R_k$ est le *corps* (*body*) de la règle. Toute variable qui apparaît dans la tête d'une règle doit apparaître au moins une fois dans le corps de cette règle. Dans un programme datalog, les *prédicats extensionnels* correspondent aux prédicats qui ne sont jamais en tête de règle alors que les *prédicats intentionnels* sont ceux qui apparaissent au moins une fois à la tête d'une règle.

Un programme $\mathcal{P}$ est un *programme datalog monadique* si tous les prédicats intentionnels sont unaires, c'est-à-dire, de la forme $H(x)$ où x est une variable. Soit $\mathcal{P}$ un programme datalog avec les prédicats extensionnels $R_1, \ldots, R_m$ et les prédicats intentionnels $H_1 \ldots H_l$. Soit $T = \langle D, R_1^T, \ldots, R_m^T \rangle$ une structure où chaque prédicat R_i^T, d'arité $n_i \geq 1$, est interprété comme $R_i^T \subseteq D^{n_i}$. La sémantique de $\mathcal{P}$ sur T (c'est-à-dire, $\mathcal{P}(T)$) est définie par la théorie des modèles, équivalente à la sémantique obtenue par le plus petit point fixe de l'opérateur de conséquence immédiate [AHV95, CGT90]. En suivant la rapide explication donnée dans [Lib06], nous pouvons dire que cet opérateur prend une structure $\mathcal{H}' = \langle D, H_1', \ldots, H_l' \rangle$ comme entrée et produit une nouvelle structure $\mathcal{H}'' = \langle D, H_1'', \ldots, H_l'' \rangle$ telle que un n-uplet u est dans H_i'' s'il est dans H_i' ou s'il peut être déduit par une règle dans $\mathcal{P}$ en utilisant les n-uplets stockés dans les relations extensionnelles et ceux dérivés pendant le calcul de $\mathcal{H}'$.

Une *requête datalog monadique* est une paire $(\mathcal{P}, H)$ où P est un programme datalog monadique et H est un prédicat intentionnel. La valeur de H dans $\mathcal{P}(T)$ est le résultat du programme sur T. Si le programme $\mathcal{P}$ représente un schéma XML, H peut être un prédicat spécial (par exemple un prédicat *valid*) qui est vrai seulement quand la racine de l'arbre T est évaluée correctement. Cela nous permet de construire un validateur de schéma comme un programme datalog monadique. En effet, dans [dHM07] (chapitre 5), notre but est de représenter des schémas XML par des programmes datalog qui, appliqués sur les arbres XML, vérifient la validité des documents XML par rapport à un schéma donné. Le programme datalog qui représente le schéma est une requête booléenne avec un prédicat spécial *valid* évalué sur un arbre. Selon le résultat énoncé par le théorème ci-dessous, l'arbre XML est décrit par les prédicats extensionnels *root*, $leaf$, FC (*first child*), LC (*last child*) et NS (*next-sibling*) qui représentent les relations décrivant la structure d'un arbre (similaire à la définition 2.1.3).

Théorème 2.1.2 [GK02] : *Une requête unaire sur des arbres d'arité variable est MSO-definissable ssi elle est définissable en datalog monadique sur les prédicats extensionnels root, $leaf$, FC, LC, NS et P_a pour $a \in \Sigma$. Chaque requête datalog monadique $(\mathcal{P}, H)$ peut être évaluée sur un arbre T en temps $O(|\mathcal{P}|.|T|)$.* □

Remarquer que de façon similaire à la définition 2.1.3 nous supposons les relations suivantes comme correspondant aux prédicats cités par le théorème 2.1.2.

- *root* est une relation unaire telle que $root = \{\varepsilon\}$.
- $leaf$ est une relation unaire telle que $leaf = \{u \in D \mid \neg\exists u' \in D$ tel que $u' = u.i$ pour $i \in \mathbb{N}\}$.
- La relation *first child* est $u \prec_{fc} u.0$, pour $u, u.0 \in D$.
- La relation *last child* est $u \prec_{lc} u' \Leftrightarrow (u' = u.i) \wedge (u.i \in D) \wedge u.(i+1) \notin D$ pour $i \in \mathbb{N}$.
- La relation *next-sibling* est celle de la définition 2.1.3

Dans ce mémoire nous allons considérer l'utilisation des programmes datalog monadique pour la validation des contraintes de schéma. Le chapitre 5 présente un algorithme de traduction d'un schéma en programme datalog monadique. En effet, dans nos recherches nous avons été particulièrement intéressés par la validation incrémentale des contraintes. Dans nos travaux nous avons utilisé les automates d'arbres comme outil de validation des contraintes de schémas (chapitre 4), mais, plus récemment, nous avons aussi proposé un validateur en forme d'un programme datalog monadique (section 5.5).

2.2 Langages, grammaires et automates d'arbres

Alors qu'une grammaire hors contexte (CFG) ([HMU01]) engendre des mots, une grammaire d'arbres engendre des arbres ([CDG+02]). La plupart des travaux sur XML considère les documents XML comme des arbres appartenant à un langage régulier d'arbres, c'est-à-dire, un langage engendré par une grammaire régulière d'arbres. La définition d'une grammaire (régulière) d'arbres pour des arbres d'arité variable est inspirée de sa version pour les arbres d'arité fixe ([CDG+02]) en permettant que la partie droite d'une

règle de production soit une expression régulière ([MLMK05]).

Définition 2.2.1 - Grammaire d'arbres : Une grammaire (régulière) d'arbres (RTG) est un n-uplet $\mathcal{G} = (\Sigma, NT, P, S)$ où Σ est un ensemble fini des symboles (les terminaux), NT est un ensemble fini de non terminaux, disjoint de Σ ; $S \subset NT$ est un ensemble de symboles initiaux et P est un ensemble de règles de productions de la forme $A \rightarrow a[E]$ où $A \in NT$, $a \in \Sigma$ et E est une expression régulière[6] sur NT. Nous notons $\mathcal{L}(\mathcal{G})$ le langage d'arbres engendré par une grammaire d'arbres $\mathcal{G}$. □

Comme dans [MLMK05], nous considérons les grammaires dans la forme normale définie dans [LMM00]. Ainsi, les règles du type $X \rightarrow a[E_1]$ et $X \rightarrow a[E_2]$, avec le même non-terminal à gauche et le même terminal à droite, sont traduites en une seule règle $X \rightarrow a[E_1 + E_2]$. De plus, si nous avons deux règles $A \rightarrow a[E_1]$ et $A \rightarrow b[E_2]$, la grammaire sera réécrite en remplaçant, disons, $A \rightarrow a[E_1]$ par $A1 \rightarrow a[E_1]$, en laissant $A \rightarrow b[E_2]$ e en remplaçant tout non terminal A qui apparaît à droite d'une règle de production par $A1 + A$.

L'ensemble des langages engendré par les grammaires d'arbres constitue la classe de langages réguliers d'arbres (RTL). Cette classe de langages correspond exactement aux langages reconnus par les automates d'arbres. La définition des automates d'arbres dans le contexte des arbres d'arité non borné ([BMW01, BW98b]) est une extension de celle introduite pour les arbres d'arité fixe, en particulier les arbres binaires ([CDG$^+$02]). Les automates d'arbres sont, en général, classés en *ascendants* (*bottom-up*), dont l'exécution commence par les feuilles de l'arbre et considère toujours les fils avant de passer à leur père, ou *descendants* (*top-down*), dont l'exécution commence par la racine et considère le père avant de traiter les fils. Dans nos travaux nous considérons les automates ascendants déterministes ou non-déterministes.

Définition 2.2.2 - Automate d'arbres ascendant (*bottom-up finite tree automaton*) : Un automate d'arbres sur un alphabet Σ est un n-uplet $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ où Q est un ensemble fini d'états disjoint de Σ, $Q_f \subseteq Q$ est un ensemble d'états finaux et Δ est un ensemble fini de transitions de la forme $\delta(a, E) = q$ où $a \in \Sigma$, E est une expressions régulière sur Q et $q \in Q$. Nous notons $\mathcal{L}(\mathcal{A})$ le langage d'arbres reconnu par un automate d'arbres $\mathcal{A}$. □

La définition standard d'un automate d'arbres déterministe est obtenue par l'imposition d'une contrainte sur l'ensemble de transitions : pour tous les symboles a et tous les états q_1 , q_2 s'il existe une transition $\delta(a, E_1) = q_1$ et $\delta(a, E_2) = q_2$ alors $L(E_1) \cap L(E_2) = \emptyset$. La définition 2.2.2 ne fait pas cette restriction et correspond donc à un automate d'arbres non-déterministe.

Il est intéressant de remarquer que, dans le cadre des grammaires nous disons qu'un langage d'arbres est engendré par une grammaire en commençant par un symbole dans S dans une vision *top-down*. Dans le cadre des automates nous parlons de la reconnaissance par les automates car nous partons d'un terme du langage (un arbre) et nous essayons de le réduire à un état final. Il s'agit ainsi d'une vision *bottom-up*. Cette notion est formalisée, dans la suite par les définitions d'interprétation d'un arbre par rapport à une grammaire et de l'exécution d'un automate d'arbres.

Définition 2.2.3 - Interprétation d'un arbre T par rapport à une grammaire ([MLMK05]) : Une interprétation d'un arbre $T = (D, t)$ par rapport à une grammaire d'arbres $\mathcal{G} = (\Sigma, NT, P, S)$ est un arbre $R_{\mathcal{I}} = (D, \mathcal{I})$ où $\mathcal{I}$ est une fonction qui associe chaque position $p \in D$ à un non terminal dans NT en respectant les propriétés suivantes :

1. $\mathcal{I}(\epsilon) \in S$ et
2. pour chaque position p et ses enfants $p.0, \ldots p.(n-1)$ dans D il existe une règle de production $X \rightarrow a[E]$ dans $\mathcal{G}$ telle que
   (a) $\mathcal{I}(p)$ est X
   (b) le terminal (le label) associé à p est a (c'est-à-dire, $t(p) = a$) et
   (c) $\mathcal{I}(p.0). \ldots . \mathcal{I}(p.(n-1))$ est un mot de $L(E)$.

Un arbre T est *valide* par rapport à une grammaire d'arbres $\mathcal{G}$ s'il existe une interprétation de T par rapport à $\mathcal{G}$. Un ensemble d'arbres est un *langage régulier* si, pour une grammaire d'arbres $\mathcal{G}$, tous les arbres dans cet ensemble, et seulement eux, sont valides par rapport à $\mathcal{G}$. □

La définition de l'interprétation d'un arbre par rapport à une grammaire a son homologue dans le cadre des automates d'arbres où nous parlons d'exécution d'un automate sur un arbre donné. Remarquer

[6] Étant donné un alphabet $\Sigma = \{a_1, \ldots, a_n\}$, nous rappelons que l'ensemble des expressions régulières sur Σ est inductivement défini par : $E ::= \emptyset \mid \epsilon \mid a_i \mid E + E \mid E.E \mid E^+ \mid E^* \mid E? \mid (E)$.

que dans ce cas, au lieu des non terminaux nous avons les états.

Définition 2.2.4 - Exécution de $\mathcal{A}$ sur un arbre T : Soit $T = (D, t)$ un arbre et soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un automate d'arbres (définition 2.2.2). L'exécution de $\mathcal{A}$ (*run*) sur T est un arbre $R_{\mathcal{A}} = (D, r)$ où r est une fonction qui associe chaque position p du domaine D à un ensemble d'états $\mathcal{Q}_p$ tel que pour tous les états $q \in \mathcal{Q}_p$ les conditions ci-dessous sont respectées :

1. $t(p) = a \in \Sigma$
2. Il existe une règle de transition $\delta(a, E) = q$ dans Δ.
3. Il existe un mot $w = q_0 \dots q_{n-1}$ dans $L(E)$ tel que $q_0 \in \mathcal{Q}_0 \dots q_{n-1} \in \mathcal{Q}_{n-1}$ où $\mathcal{Q}_0 \dots \mathcal{Q}_{n-1}$ sont les ensembles d'états associés à chaque fils de p.

Une exécution r est *réussie* si $r(\epsilon) = \mathcal{Q}$ et $\mathcal{Q} \cap Q_f \neq \emptyset$. Un arbre T est *valide* s'il existe une exécution réussie sur T. Un arbre T est *localement valide*, si $r(\epsilon)$ est associé à un ensemble $\mathcal{Q}$ tel que $\mathcal{Q} \subset Q$ mais $\mathcal{Q} \cap Q_f$ peut être vide. □

Dans la définition 2.2.4, si nous considérons que pour tout $0 \leq i \leq (n-1)$ nous avons $q_i^j \in \mathcal{Q}_{p.i}$, la condition 3 demande que $L(E) \cap L(E_{aux}) \neq \emptyset$, où $E_{aux} = (q_0^0 | q_0^1 | \dots | q_0^{k_0}) \dots (q_{n-1}^0 | q_{n-1}^1 | \dots | q_{n-1}^{k_{n-1}})$ est une expression régulière représentant l'ensemble des mots composés par les états associés aux n fils de p (en supposant que le nombre d'états dans chaque $\mathcal{Q}_{p.i}$ est $k_i + 1$).

Classification des langages réguliers d'arbres

Nous rappelons la classification des langages d'arbres proposée dans [MLMK05] en prenant en compte deux sous classes des langages réguliers d'arbres. Nous présentons les définitions dans les contextes des grammaires ([MLMK05]) et des automates ([BCH+07]), en faisant un parallèle entre eux.

Nous commençons en rappelant la définition des *non terminaux en concurrence.* Étant donnée une grammaire $\mathcal{G}$, deux non terminaux différents A et B de $\mathcal{G}$ sont en concurrence entre eux s'il existe deux règles de productions en $\mathcal{G}$ de la forme $A \rightarrow a[E_1]$ et $B \rightarrow a[E_2]$. Dans le contexte des automates nous parlons des *états en concurrence.* Étant donné un automate d'arbres $\mathcal{A}$, deux états différents q_A et q_B de $\mathcal{A}$ sont concurrents s'il existe deux règles de transition δ_1 et δ_2, telles que $\delta_1(a, E_1) = q_A$ et $\delta_2(a, E_2) = q_B$.

Une *grammaire d'arbres locale* (LTG) est une grammaire régulière d'arbres sans non terminaux en concurrence . Un ensemble d'arbres est un *langage d'arbres local* (LTL) si, pour une LTG $\mathcal{G}$, tous les arbres de cet ensemble, et seulement eux, sont valides par rapport à $\mathcal{G}$. Autrement dit, un LTL est l'ensemble des arbres reconnus par un automate d'arbres qui n'admet pas la concurrence entre ses états.

Une *grammaire d'arbres à type unique* ($STTG$) est une grammaire régulière d'arbres où les symboles de démarrage ne sont pas en concurrence et dans ses règles de production, les non terminaux dans une même expression régulière ne sont pas en concurrence. Un ensemble d'arbres est un *langage d'arbres à type unique* ($STTL$) si, pour une $STTG$ $\mathcal{G}$, tous les arbres de cet ensemble, et seulement eux, sont valides par rapport à $\mathcal{G}$. Autrement dit, un $STTL$ est le langage reconnu par un automate d'arbres qui satisfait les contraintes suivantes : (i) deux états dans Q peuvent être en concurrence mais ils ne doivent pas apparaître dans la même expression régulière E d'une règle de transition et (ii) l'ensemble des états finaux de Q_f est un singleton.

Exemple 2.2.1 Soient trois automates $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$ construits pour reconnaître les langages engendrés par les grammaires d'arbres $\mathcal{G}_1$, $\mathcal{G}_2$ et $\mathcal{G}_3$, respectivement. Chaque grammaire $\mathcal{G}_i$ est définie par $\mathcal{G}_i = (\Sigma, NT, P_i, S)$, alors que chaque automate $\mathcal{A}_i = (Q, \Sigma, Q_f, \Delta_i)$. Les terminaux sont en minuscule alors que les non terminaux commencent par une lettre majuscule. Les trois ensembles de règles de production possèdent les règles $A \rightarrow a[Data]$ où $a \in \{name, add, phone, number\}$ et $Data \rightarrow data[\epsilon]$ en plus des règles spécifiques. La même supposition est faite pour les trois automates.

P_1		Δ_1	
Dir	$\rightarrow directory[Person^*]$	$\delta(directory, (q^*_{person})) =$	q_{dir}
$Person$	$\rightarrow student[DirA \mid DirB])$	$\delta(student, (q_{dirA} \mid q_{dirB})) =$	q_{person}
$Person$	$\rightarrow professor[DirB]$	$\delta(professor, (q_{dirB})) =$	q_{person}
$DirA$	$\rightarrow direction[Name.Number?.Add?]$	$\delta(direction, (q_{name}.q_{number}?.q_{add}?)) =$	q_{dirA}
$DirB$	$\rightarrow direction[Name.Add?.Phone^*]$	$\delta(direction, (q_{name}.q_{add}?.q^*_{phone})) =$	q_{dirB}

P_2		Δ_2	
Dir	$\rightarrow directory[Person^*]$	$\delta(directory, (q^*_{person})) =$	q_{dir}
$Person$	$\rightarrow student[DirA])$	$\delta(student, (q_{dirA})) =$	q_{person}
$Person$	$\rightarrow professor[DirB]$	$\delta(professor, (q_{dirB})) =$	q_{person}
$DirA$	$\rightarrow direction[Name.Number.Add?]$	$\delta(direction, (q_{name}.q_{number}.q_{add}?) =$	q_{dirA}
$DirB$	$\rightarrow direction[Name.Add?.Phone^*]$	$\delta(direction, (q_{name}.q_{add}?.q^*_{phone})) =$	q_{dirB}

P_3		Δ_3	
Dir	$\rightarrow directory[Student^*.Professor^*]$	$\delta(directory, (q^*_{stud}.q^*_{prof})) =$	q_{dir}
$Student$	$\rightarrow student[Name.Number.Add?]$	$\delta(student, (q_{name}.q_{number}.q_{add}?)) =$	q_{stud}
$Professor$	$\rightarrow professor[Name.Add?.Phone^*]$	$\delta(professor, (q_{name}.q_{add}?.q^*_{phone})) =$	q_{prof}

Selon les définitions précédentes, nous pouvons conclure que $\mathcal{G}_3$ est une LTG, $\mathcal{G}_2$ est une $STTG$ qui n'est pas LTG ($DirA$ et $DirB$ sont en concurrence) alors que $\mathcal{G}_1$ est une RTG qui n'est pas $STTG$ ($DirA$ et $DirB$ sont en concurrence et font partie d'une même expression régulière). □

Pouvoir d'expression et propriétés des langages réguliers d'arbres

Le pouvoir d'expression des trois classes de langages est étudié dans [MLMK05] et peut être résumé par l'inclusion $LTL \subset STTL \subset RTL$.

Les propriétés de ces trois classes de langages ne sont pas toujours les mêmes ([MLMK05]). Ainsi, sur la question de l'unicité des interprétations nous savons que, par rapport aux $STTG$ ou LTG, un arbre a au plus une interprétation. Par contre, un arbre peut avoir plusieurs interprétations par rapport à une RTG. Comme une LTG interdit des non terminaux en concurrence, une seule règle de production peut être utilisée pour un nœud. Cela équivaut à dire qu'un automate construit pour des LTL est déterministe. Une $STTG$ interdit des non terminaux en concurrence à la racine. Cela nous permet de résoudre les éventuelles concurrences sur les autres nœuds : les concurrences des non terminaux sur un nœud p sont résolues en utilisant les informations concernant le père de p. Cette situation spéciale des $STTL$ et LTL nous permet de construire des algorithmes de validation incrémentale sans l'utilisation de structures de stockage auxiliaires. Néanmoins, ces structures sont nécessaires pour les RTL (non $STTL$).

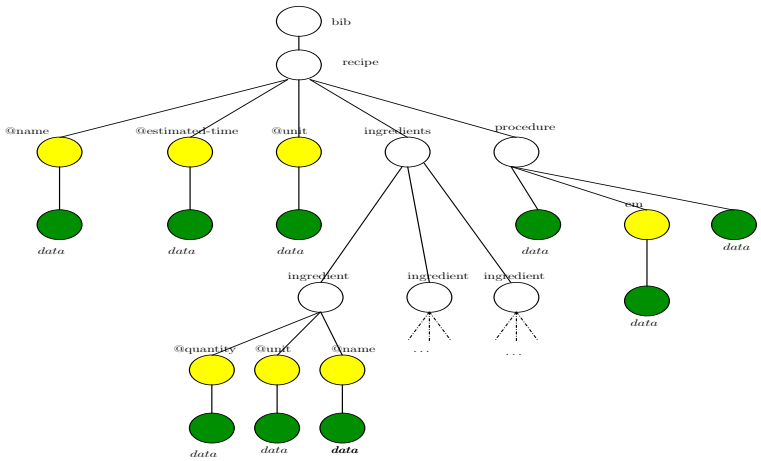
Sur la question de la fermeture booléenne. Les LTL et les $STTL$ ne sont pas fermées par union et différence mais sont fermées par intersection. Les RTL sont fermées par union, différence et intersection. Nous rappelons aussi que le test du vide (c'est-à-dire, est-ce qu'un langage est vide ?) et le test d'appartenance (c'est-à-dire, est-ce qu'un arbre appartient au langage ?) sont décidables pour les langages réguliers.

Langages de schéma XML

Les langages de schéma pour XML proposés aujourd'hui peuvent être représentés par les grammaires d'arbres de la définition 2.2.1 (ce que nous ferons dans ce mémoire). Néanmoins, ils sont parfois présentés par une variation de cette définition où il n'y a pas de distinction entre les symboles terminaux et non terminaux ([BMW01, MLMK05]).

Dans [MLM01, MLMK05] une classification des principaux langages est proposée. Ainsi une DTD est une LTG, le W3C XML Schema (XSD) est une $STTG$ et RELAX NG est une RTG. Remarquer que la non unicité de l'interprétation pour les RTL n'est pas forcément une faiblesse, comme il est expliqué dans [MLMK05]. Les types dans les langages de programmation XML (plusieurs langages de programmation ont été développés comme, par exemple XDuce[7] and JWIG[8] sont *set-theoretic*([KSS03]). Autrement dit, un type A représente un ensemble $\mathcal{L}(A)$ et un élément est du type A ssi il appartient au langage $\mathcal{L}(A)$. Les types *set-theoretic* ne demandent pas l'unicité des interprétations (c'est-à-dire, un élément peut avoir plusieurs types). Néanmoins il faut rappeler que des approches différentes existent. Dans XQuery, par exemple, un document XML est stocké avec son interprétation (construite pendant la validation).

[7] XDuce : `http ://xduce.sourceforge.net/index.html`
[8] JWIG : `http ://www.brics.dk/JWIG/manual/introduction.html`




```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bib SYSTEM "recipeBib.dtd">
<bib>
 <recipe name="My first cake" estimated-time="1" unit="h" >
  <ingredients>
    <ingredient quantity="200" unit="g" name="flour"/>
    <ingredient quantity="100" unit="g" name="sugar"/>
    <ingredient quantity="2" unit="dl" name="milk"/>
  </ingredients>
  <procedure>Throw all the ingredients in a bowl and stir
   <em>vigorously</em>. Pour resulting batter into a pan and bake
  for thirty minutes.
  </procedure>
 </recipe>
</bib>
```

FIG. 2.2: Exemple recettes([KSS03]). Document XML représenté par un arbre (représentation adoptée dans [BDHL03]) et par un texte. Sur l'arbre les attributs sont signalés par @ et les nœdus *data* sont ceux associés à des valeurs.

Les avis sur l'utilisation des langages de schéma XML peuvent différer. L'unicité de l'interprétation peut être considérée comme un avantage pour certains (ce qui privilégie LTG et $STTG$) alors que la non unicité n'est pas un problème pour ceux qui défendent plus de pouvoir d'expression. Selon [JNV04], en termes de besoin pour les demandes pratiques actuelles, les DTDs semblent être suffisantes pour représenter la grande majorité des contraintes de schéma souhaitées.

Quelques précisions sur le traitement des attributs et la représentation des données

Faisons une comparaison entre l'arbre d'arité variable (c'est-à-dire, l'arbre XML selon la définition 2.1.1) et un document (texte) XML (figure 2.2). Les noms des éléments et des attributs dans un document XML sont traduits dans des labels associés aux positions de l'arbre. Néanmoins, les données (valeurs) n'ont pas été considérées dans la définition 2.1.1. Dans nos travaux, nous supposons, en effet, l'existence d'un label spécifique *data*. Toute position dont le label est *data* est associée à une valeur dans un domaine infini DOM. Pour accéder à cette valeur, nous proposons une fonction $value$ définie sur les nœuds *data*. Les nœuds *data* sont toujours des feuilles. Dans ce mémoire les nœuds *data* seront considérés seulement pendant la description de nos travaux sur les contraintes d'intégrité car, dans ce contexte, nous avons besoin de faire des comparaisons sur les valeurs. Bref, même si un document XML peut toujours être converti dans la représentation abstraite que nous utilisons, pour que la réciproque soit vraie certaines conditions doivent être respectées (différent types de nœuds, nœuds du type *data* toujours feuilles, nœuds attributs précédant les éléments, etc). Nous référençons [KSS03] pour une discussion à ce sujet et [BDHL03] pour des précisions sur le modèle que nous avons utilisé dans la plupart de nos travaux.

Une autre remarque importante concerne les attributs. Les définitions de l'automate d'arbres et de son exécution (définitions 2.2.2 et 2.2.4) sont des versions simplifiées de celle utilisée dans nos travaux où les attributs des documents XML sont pris en compte. Dans nos travaux (par exemple [BH03, BHM03, BCH$^+$07]), nous avons supposé que les premiers nœuds fils de p pouvaient être des nœuds attributs. Ces nœuds méritent un traitement spécial puisqu'ils doivent être considérés comme des éléments d'un ensemble où l'ordre n'est pas important, contrairement aux fils du type élément où l'ordre est pris en compte. Pour traiter ces différents types de nœuds, nous avons proposé un automate d'arbres étendu avec des fonctions de transitions de la forme $\delta(a, S, E) = q$ où $S = (S_{compusory}, S_{optional})$, et $S_{compusory}$ et $S_{optional}$ sont des ensembles d'états qui représentent, respectivement, les attributs obligatoires et optionnels. L'exécution de cet automate étendu correspond à la définition 2.2.4 avec l'ajout de conditions concernant la vérification des attributs. Dans ce mémoire nous laissons de coté les détails concernant le traitement des attributs.

Chapitre 3

Mises à jour et maintenance des documents XML

Dans ce chapitre, nous considérons les détails de nos opérations de mises à jour. D'abord nous présentons la sémantique des mises à jour élémentaires et ensuite la notion de mises à jour multiples. Le concept de liste de mises à jour non contradictoire est introduit pour prouver que les mises à jour multiples peuvent se réduire à une composition commutative de mises à jour élémentaires. Ce résultat est inédit et vient compléter les travaux de recherches XML que nous présentons ici. Nous finissons le chapitre par une discussion générale sur la maintenance de contraintes dans une base de données XML lors de la présence de mises à jour et sur certains travaux liés.

3.1 Préliminaires

Soit un arbre $T = (D, t)$. Pour spécifier nos mises à jour sur T, nous introduisons les ensembles de positions ci-dessous :

- L'ensemble *frontière d'insertion* (*insert frontier*) de T :
  $fr^{ins}(T) = \{u.i \notin D \mid u \in D \wedge i \in \mathbb{N} \wedge [(i = 0) \vee ((i \neq 0) \wedge u.(i-1) \in D)]\}$.
  Pour l'arbre vide T, $fr^{ins}(T) = \{\epsilon\}$. De façon intuitive, la frontière d'insertion contient les positions qui ne sont pas dans l'arbre original, mais où une insertion est possible.

- Les ensembles $DelPos_p$, $ShiftRightPos_p$, et $ShiftLeftPos_p$, définis par rapport à une position $p \neq \epsilon$. Soit $p = u.i$, où $p \in D$, $i \in \mathbb{N}$ et $u \in \mathbb{N}^*$. Soit $n+1$ le nombre de fils (*fan-out*) du père de p.
  - $DelPos_p = \bigcup_{k=i}^{n} \{v \mid v \in D,\ v = u.k.u' \text{ et } u' \in \mathbb{N}^*\}$.
  - $ShiftRightPos_p = \bigcup_{k=i}^{n} \{v \mid v = u.(k+1).u',\ u.k.u' \in D \text{ et } u' \in \mathbb{N}^*\}$.
  - $ShiftLeftPos_p = \bigcup_{k=i+1}^{n} \{v \mid v = u.(k-1).u',\ u.k.u' \in D \text{ et } u' \in \mathbb{N}^*\}$.

L'ensemble $DelPos_p$ correspond aux positions qui doivent être modifiées dû à une insertion ou à une suppression à la position p de l'arbre. $ShiftRightPos_p$ (respectivement, $ShiftLeftPos_p$) est l'ensemble des positions obtenues par un déplacement (*shift*) à droite (respectivement, à gauche) des positions originales suite à une insertion (respectivement, une suppression) sur p.

RAPPEL : Dans nos travaux (et parfois dans ce mémoire) nous utilisons le symbole $\preceq$ de façon équivalente à $\prec_{ch}^*$; alors que $\prec$ indique un préfixe strict, c'est-à-dire, si $p \prec p'$ alors $p \preceq p' \wedge p \neq p'$. Rappelons aussi que $\prec_{ch}$ représente la relation *child* (Définition 2.1.3).

Définition 3.1.1 - Arbres isomorphes : Soient $T = (D, t)$ et $T' = (D', t')$ deux arbres. Une fonction *bijective* $h : D \rightarrow D'$ est un isomorphisme de l'arbre T vers l'arbre T' si les conditions suivantes sont vérifiées :

1. La racine de T est associée à la racine de T', c'est-à-dire, $h(\epsilon) = \epsilon$ et $t(\epsilon) = t'(\epsilon)$.
2. $t(v) = t'(h(v))$ pour toute position $v \in D$.

3. Pour chaque couple (v, u) tel que $v, u \in D$ et $v \prec_{ch} u$, nous avons $h(v) \prec_{ch} h(u)$ et pour chaque couple (v', u') tel que $v', u' \in D'$ et $v' \prec_{ch} u'$, nous avons $h^{-1}(v') \prec_{ch} h^{-1}(u')$. □

3.2 Mises à jour

La définition ci-dessous introduit les opérations de mises à jour élémentaires que nous utilisons dans notre approche.

Définition 3.2.1 - Mises à jour élémentaires : Une opération de mise à jour *upd* est un n-uplet (op, p, Γ) où :
(i) *op* est le nom de l'opération, $op \in \{insert, delete, replace\}$;
(ii) $p = u.i$ ($u \in \mathbb{N}^*$ et $i \in \mathbb{N}$) est une position de mise à jour définie selon l'opération *op* par rapport au domaine de l'arbre sur lequel l'opération sera appliquée et
(iii) $\Gamma = (D_\Gamma, \tau)$ est un arbre.

Étant donné un arbre non vide $T = (D, t)$, une mise à jour *upd* est une fonction partielle qui transforme T dans un nouvel arbre $T' = (D', t')$. Nous notons $T \xrightarrow{upd} T'$ ou $upd(T) = T'$.

- Si $upd = (insert, p, \Gamma)$ où $p \in (D \cup fr^{ins}(T)) \setminus \{\epsilon\})$ et $\Gamma \neq \emptyset$ alors
  - $D' = [D \setminus DelPos_p] \cup ShiftRightPos_p \cup \{p.v \mid v \in D_\Gamma)\}$
  - $\begin{cases} t'(w) = t(w), & \forall w \in (D \setminus DelPos_p). \\ t'(u.(k+1).u') = t(u.k.u') & \text{pour chaque } u.(k+1).u' \in ShiftRightPos_p \\ & \text{où } u' \in \mathbb{N}^* \text{ et } i \leq k \leq n, \\ & \text{et } n+1 \text{ est le } \textit{fan-out} \text{ du père de } p. \\ t'(p.v) = \tau(v) & \text{pour chaque } v \in D_\Gamma \end{cases}$

  Remarque : L'insertion à la position ϵ d'un arbre non vide n'est pas définie.

- Si $upd = (delete, p, \Gamma_{empty})$ où $p \in (D \setminus \{\epsilon\})$ et Γ_{empty} est un arbre vide, alors
  - $D' = [D \setminus DelPos_p] \cup ShiftLeftPos_p$
  - $\begin{cases} t'(w) = t(w) & \text{pour chaque } w \in (D \setminus DelPos_p) \\ t'(u.(k-1).u') = t(u.k.u') & \text{pour chaque } u.(k-1).u' \in ShiftLeftPos_p \\ & \text{où } u' \in \mathbb{N}^* \text{ et } (i+1) \leq k \leq n, \text{ où } n+1 \\ & \text{est le nombre de fils } (\textit{fan-out}) \text{ de } p. \end{cases}$

- Nous définissons $T \xrightarrow{(delete, \epsilon, \Gamma_{empty})} (\{\epsilon\}, \{(\epsilon, \lambda)\})$.

- Si $upd = (replace, p, \Gamma)$ où $p \in D$ et $\Gamma \neq \emptyset$, alors
  - $D' = [D \setminus \{v \mid v \in D \wedge v = p.u'\}] \cup \{p.v \mid v \in D_\Gamma\}$ où $u' \in \mathbb{N}^*$
  - $\begin{cases} t'(w) = t(w) & \forall w \in D \text{ et } w \neq p.u' \\ t'(p.v) = \tau(v) & \forall v \in D_\Gamma \end{cases}$

Quand T est un arbre vide, les opérations d'insertion et de remplacement donnent comme résultat l'arbre Γ. En d'autres termes, nous avons $(\{\epsilon\}, \{(\epsilon, \lambda)\}) \xrightarrow{(insert, \epsilon, \Gamma)} \Gamma$ et $(\{\epsilon\}, \{(\epsilon, \lambda)\}) \xrightarrow{(replace, \epsilon, \Gamma)} \Gamma$. La suppression sur un arbre vide, ne change rien, c'est-à-dire $(\{\epsilon\}, \{(\epsilon, \lambda)\}) \xrightarrow{(delete, \epsilon, \Gamma)} (\{\epsilon\}, \{(\epsilon, \lambda)\})$. □

Après avoir considéré la validation incrémentale des documents XML soumis à des mises à jour simples (dans [BH03]), nous avons proposé une extension de nos méthodes de validation (par rapport aux schémas et aux contraintes d'intégrité) dans [BCH+07], en considérant une suite de mises à jour. Nous considérons ainsi des mises à jour multiples où une liste d'opérations de mises à jour est traitée comme une unique transaction. Nous adoptons ainsi les principes de [BBFV05, SHS04] et la validité d'un document est établie seulement après que *toute* la suite des opérations de mises à jour soit considérée. Notre validateur considère que cette suite doit être *non contradictoire* ; concept introduit par la définition ci-dessous.

Définition 3.2.2 - Mises à jour contradictoires : Deux opérations de mises à jour $upd_1 = (op_1, p_1, \Gamma_1)$ et $upd_2 = (op_2, p_2, \Gamma_2)$ sont *contradictoires* si et seulement si une des conditions suivantes est vérifiée :

1. $p_1 = \epsilon$ ou $p_2 = \epsilon$.

2. Les opérations indiquent une suppression ou un remplacement sur une même position. En d'autres termes, $upd_1 \neq upd_2$, $p_1 = p_2$; $op_1 \in \{delete, replace\}$ et $op_2 \in \{delete, replace\}$.
3. Une opération indique une suppression ou un remplacement sur une position alors que l'autre opération indique une autre opération sur un descendant (strict). En d'autres mots, $p_1 \prec p_2$, $op_1 \in \{delete, replace\}$ et $op_2 \in \{insert, delete, replace\}$ ou $p_2 \prec p_1$, $op_2 \in \{delete, replace\}$ et $op_1 \in \{insert, delete, replace\}$

Une liste de mises à jour est *non contradictoire*, s'elle ne contient pas d'opérations de mises à jour contradictoires. □

Étant donnée une liste de mises à jour souhaitées par un utilisateur, nous considérons qu'un pré-traitement transforme cette liste en liste non contradictoire. Nous ne proposons pas un pré-traitement spécifique. En cas de contradiction, il peut demander aux utilisateurs de faire un choix ou établir des priorités entre les opérations.

Il est intéressant de remarquer, dans le cas 1 de la définition 3.2.2, qu'une liste de mises à jour contenant une opération sur la position ϵ est considérée comme contradictoire. Ainsi, les opérations (possibles) sur la racine d'un arbre donné doivent être faites indépendamment des autres mises à jour. Cela s'explique car, en effet, une mise à jour sur ϵ représente une transformation majeure, qui change tout l'arbre (indiquant qu' un nouveau document est pris en compte)[1]. Remarquer aussi que, mise à part les insertions, le cas 1 est un cas particulier de la condition 3 de la définition 3.2.2.

Soit un document XML et une liste des mises à jour souhaitées par l'utilisateur. Notre algorithme de validation incrémentale prend en compte les mises à jour de la liste au fur et à mesure qu'il parcourt le document XML. Pour des questions d'efficacité, nous définissons une liste où les positions des mises à jour respectent l'ordre de lecture du document XML. La définition ci-dessous introduit la notion de mise à jour multiple et montre comment elle se décompose en mises à jour élémentaires. Il est important de remarquer que l'application de chaque mise à jour élémentaire provoque un changement sur l'arbre ainsi que sur la liste de mises à jour.

Dans la définition ci-dessous, nous utilisons des fonctions standards sur une liste, à savoir $Head(L)$ qui récupère la mise à jour élémentaire qui est à la tête d'une liste L et $Body(L)$ qui nous donne la liste L sans sa tête. Pour simplifier la notation, nous supposons l'existence de la fonction $ShiftRight$ (respectivement $ShiftLeft$) qui reçoit une liste de mises à jour et calcule une autre liste de mises à jour en effectuant un "*shift*" à droite (respectivement, à gauche) des positions de mises à jour affectées par la mise à jour précédente (de manière similaire à ce que propose la définition de $ShiftRightPos_p$ et, respectivement, $ShiftLeftPos_p$).

Définition 3.2.3 - Mises à jour multiples : Une liste de mises à jour L est une suite *non contradictoire* de mises à jour élémentaires (respectant les conditions établies par la Définition 3.2.1). Les n-uplets (op, p, Γ) dans L sont ordonnés par rapport aux positions p et selon l'ordre de lecture du document.

Étant donné un arbre non vide T, une mise à jour multiple est une composition des fonctions qui transforme T dans un nouvel arbre T'. Nous notons $T \xrightarrow{(L)} T'$ pour indiquer $T = T_0 \xrightarrow{Head(L_1)} T_1 \xrightarrow{Head(L_2)} T_2 \ldots \xrightarrow{Head(L_n)} T_n = T'$ où $L_1 = L$, $Body(L_n) = []$ et, pour $2 \leq i \leq n$, chaque liste $L_i = Shift(L_{i-1})$.

La fonction $Shift$ est définie de la façon suivante :

$$Shift(L) = \begin{cases} ShiftLeft(Body(L)) & \text{Si } Head(L) = (delete, p, (\{\epsilon\}, \{\epsilon, \lambda\})) \\ ShiftRight(Body(L)) & \text{Si } Head(L) = (insert, p, \Gamma) \\ Body(L) & \text{Si } Head(L) = (replace, p, \Gamma) \end{cases}$$ □

Ainsi, une mise à jour multiple est l'application d'une liste de mises à jour sur un arbre T donné. L'application de la liste $L(= L_1)$ sur T équivaut à effectuer les étapes suivantes jusqu'à ce que la liste de mises à jour soit vide :
(i) appliquer la i-ème mise à jour élémentaire upd_i sur T_{i-1} pour obtenir T_i ;
(ii) supprimer upd_i de L_i et
(iii) corriger (en prenant en compte la mise à jour upd_i qui vient d'être effectuée) les positions des autres mises à jour encore dans L_i (pour obtenir L_{i+1}).

[1]Selon la définition 3.2.1, nous pouvons envisager les mises à jour suivantes sur ϵ : *insert* sur un arbre vide ; *delete* ou un *replace* sur un arbre quelconque. La seule opération qui n'implique pas un changement de document est $(\{\epsilon\}, \{(\epsilon, \lambda)\}) \xrightarrow{(delete, \epsilon, \Gamma)} (\{\epsilon\}, \{(\epsilon, \lambda)\})$.

Il est important de remarquer les caractéristiques particulières des insertions. Dans notre approche, une opération *insert* sur une position p n'est pas contradictoire avec une autre mise à jour (*insert*, *delete*, ou *replace*) sur une position p' telle que $p \prec_{ch}^{*} p'$. En effet, une insertion sur une position p provoque l'ajout d'un nouvel arbre en p et un "*shift*" à droite sur toutes les positions à droite de p. Le "*shift*" à droite est aussi appliqué sur la liste des mises à jour provoquant un changement sur les positions des mises à jour pas encore appliquées sur l'arbre.

Exemple 3.2.1 Soit une liste $L_1 = [(insert, 1, \Gamma_A), (insert, 1, \Gamma_B), (insert, 1, \Gamma_C)]$ appliqué sur $T = T_0$, où $t_0 = r(\tau_I, \tau_F)$ avec r le label du nœud racine. Si l'on note T' l'arbre tel que $T \xrightarrow{L_1} T'$ avec $t' = r(\tau_I, \tau_A, \tau_B, \tau_C, \tau_F)$, les étapes intermédiaires de la construction de cet arbre sont les suivantes : La mise à jour $T_0 \overset{(insert,1,\Gamma_A)}{\longrightarrow} T_1$ donne $t_1 = r(\tau_I, \tau_A, \tau_F)$ (et le domaine D_1 corrrespondant) et $ShiftRight(Body(L1))$ nous donne $L_2 = [(insert, 2, \Gamma_B), (insert, 2, \Gamma_C)]$. Ensuite, nous avons $T_1 \overset{(insert,2,\Gamma_B)}{\longrightarrow} T_2$. Le résultat T_2 contient $t_2 = r(\tau_I, \tau_A, \tau_B, \tau_F)$ et la nouvelle liste $L_3 = [(insert, 3, \Gamma_C)]$. Finalement, $T_2 \overset{(insert,3,\Gamma_C)}{\longrightarrow} T_3$ où $t_3 = r(\tau_I, \tau_A, \tau_B, \tau_C, \tau_F)$ et la liste de mises à jour devient vide ($L_4 = []$). □

Nous rappelons que les demandes de l'utilisateur portent toujours sur l'arbre d'origine, c'est-à-dire, les positions référencées par une mise à jour multiple sont toutes des positions de l'arbre original. Ainsi, les changements effectués sur la liste de mises à jour sont *internes* et ne sont pas visibles *avant* la fin de la mise à jour multiple. Notre but est de permettre à l'utilisateur d'indiquer les mises à jour souhaitées sur les positions qui existent au moment où il liste ses changements. Dans l'exemple précédent, les insertions demandées par l'utilisateur portent sur la position qu'il connaît comme étant "occupée" par le sous arbre τ_F.

Remarquer que la notion de contradiction (définition 3.2.2) a été introduite dans le but de régler les problèmes occasionnés par les différences entre les positions de mises à jour demandées par l'utilisateur (toujours sur l'arbre d'origine) et les positions de l'arbre mis à jour (calculées au fur est à mesure que les mises à jour sont considérées). Soit une liste $L = [(delete, 1, \Gamma_{empty}), (replace, 1, \Gamma_1)]$ contenant les mises à jour (contradictoires) d'un utilisateur sur un arbre T avec $t = r(\tau_A, \tau_B, \tau_C)$. La mise à jour multiple supprime le sous arbre τ_B et remplace le sous arbre τ_C par τ_1 ce qui est difficile à expliquer à l'utilisateur qui semble vouloir changer le sous arbre dont la racine est à la position 1 dans l'arbre original. La notion de mise à jour contradictoire impose un choix et évite ce genre de problème.

Dans notre approche, pour des raisons d'efficacité, si une liste de mises à jour possède des insertions et des suppressions sur la même position, le pré-traitement remplace ces opérations par une opération *replace*.

Dans ce mémoire, les restrictions imposées sur la liste de mises à jour L sont assouplies par rapport à celles imposées dans [BCH+07], sans conséquence sur les algorithmes de validation proposés et les résultats obtenus. Le théorème ci-dessous montre que si la liste de mises à jour L est non contradictoire et contient seulement des mises à jour élémentaires correctes selon la définition 3.2.1, alors, si L ne contient pas plusieurs insertions sur une même position, la mise à jour multiple est une composition commutative de fonctions. Autrement dit, le théorème fournit une condition suffisante pour la commutation de nos mises à jour.

Théorème 3.2.1 *Soit L une suite* non contradictoire *de mises à jour élémentaires (respectant les conditions établies par la Définition 3.2.1). Soient $L_1, L_2, \ldots L_m$ des suites de mises à jour construites à partir de L en changeant l'ordre des mises à jour élémentaires, c'est-à-dire, $L_1, L_2, \ldots L_m$ sont des suites de mises à jour qui ne différent entre elles que par l'ordre des mises à jour élémentaires. Soit T un arbre XML et soient les mises à jour multiples (Définition 3.2.3) $T \xrightarrow{(L_1)} T'_1$, …, $T \xrightarrow{(L_m)} T'_m$.*

Les arbres $T'_1 \ldots T'_m$, résultats des mises à jour multiples ainsi définies, sont isomorphes. Plus particulièrement, si la liste L ne contient pas plusieurs insertions sur une même position alors les arbres résultats sont identiques (c'est-à-dire $T'_1 = \cdots = T'_m = T'$).

Preuve : La preuve est faite par induction sur la longueur de L, dans l'annexe A.

3.3 Mises à jour et contraintes

Dans nos travaux de recherche nous considérons deux classes de contraintes sur les documents XML d'une base de données : *les contraintes de schéma*, définissant la structure ou le type du document XML et *les contraintes d'intégrité* exprimant des restrictions sémantiques importantes, usuellement rencontrées dans une base de données pour assurer la cohérence des informations stockées. Un problème crucial dans la gestion des bases de données concerne la maintenance des contraintes lorsque les mises à jour sont effectuées. Cette maintenance doit être, préférentiellement, incrémentale. Nos travaux de recherche sur XML se placent dans ce contexte et ont abordé le problème selon différentes optiques.

Dans le chapitre 4, nous considérons les contraintes de schéma (ou le type du document) et nous présentons une méthode de validation incrémentale en présence des mises à jour multiples. Deux approches ont été développés, à savoir : (*i*) la liste de mises à jour est refusée en cas de violation de contrainte ([BH03, BCH$^+$07]) et (*ii*) une routine de correction est lancée à chaque fois que la validation échoue pour proposer différentes versions d'un document mis de jour et valide ([BCHS06b, BCHS06a]).

Dans le chapitre 5 nous restons dans le cadre des contraintes structurales et nous abordons l'évolution de schémas d'une façon originale car c'est via les mises à jour "interdites" par la validation, mais considérées prioritaires dans le contexte, qu'une routine de transformation de schéma est déclenchée. Le rôle de la routine de transformation est d'adapter le schéma original au document mis à jour tout en préservant la validité des autres documents qui n'ont pas besoin d'être modifiés ([BDH$^+$04b, BDH$^+$04a, dHM07]).

Dans le chapitre 6 nous changeons de contexte pour prendre en compte différentes contraintes d'intégrité. Notre but étant leur validation incrémentale en présence de mises à jour multiples, nous proposons un formalisme (syntaxe) homogène qui facilite la représentation des contraintes d'intégrité et qui s'adapte bien avec notre méthode de validation ([ABH$^+$04, BCH$^+$07]).

Dans toutes nos procédures de validation (présentées dans les chapitres suivants), les mises à jour sont traitées via le parcours gauche-droite (SAX) du document XML. Cette option a été prise dans un souci d'homogénéité et d'efficacité des routines de validation. Mais selon le résultat du théorème 3.2.1 *le résultat de la mise à jour multiple serait le même si un autre parcours était adopté*. Autrement dit, une liste de mises à jour non contradictoire, fournie par un utilisateur, peut être organisée selon l'ordre de lecture du document XML sans conséquences pour la sémantique de la mise à jour. Notre procédure de mises à jour n'est pas restreinte au parcours gauche-droite de SAX car nous traitons uniquement des listes de mises à jour commutatives.

Dans [BCH$^+$07, BCHS06a] nous avons considéré des restrictions plus fortes sur les listes de mises à jour. Dans ce mémoire nous les définissons formellement et avec plus de précision. Dans [TIHW01], des primitives des mises à jour (légèrement différentes des nôtres) sont proposées ainsi que leur impleméntation sur XQuery. Le but de l'article est la définition d'un langage de mises à jour sur des documents XML et la traduction des opérations de mises à jour XML en mise à jour de la base de données relationnelle associée. Nous n'avons pas (encore) procédé à l'intégration de nos primitives de mise à jour à un langage de requêtes comme XSLT [XSL], UpdateX [SHS04] qui est intégré à XQuery [XQu] ou XUpdate[2] dont le projet semble ne pas avoir vraiment abouti. Le W3C a édité un document *XQuery Update Facility* [CFR] qui envisage des mises à jour multiples de manière similaire à celle que nous avons proposée. Les primitives citées dans XUpdate et UpdateX sont similaires aux notres. En effet, nous avons jusqu'à maintenant travaillé dans une optique où l'utilisateur manipule un document XML (ou une partie d'un document) via un éditeur qui lui donne une vision de l'arbre XML et fournit des possibilités de modifications : l'utilisateur pourrait ainsi signaler les positions (dans l'arbre XML) où il souhaiterait effectuer des insertions, des suppressions ou des remplacements. Ces changements seraient appliqués aux documents, seulement suite à une opération `commit`. Même si un prototype de cette partie interface avec l'utilisateur a été implémentée par des étudiants de master, nos thésards ont surtout travaillé sur le *back-end* des algorithmes de validation.

Notre méthode de validation incrémentale peut être vue comme une application de la sémantique *snapshot* ([BBFV05, SHS04]) où la validité d'un document est assurée seulement après avoir pris en compte toute la liste des mises à jour : toutes les opérations élémentaires dans une liste font référence aux positions de l' arbre original car même si les mises à jour élémentaires sont appliquées une à une,

[2] `http ://encyclopedia.thefreedictionary.com/xupdate`

c'est seulement à la fin de la liste que la mise à jour multiple est appliquée ou refusée. La définition 3.2.3 formalise cette idée.

Le langage XQuery ! [GRS06], extension de XQuery avec des opérations de mises à jour, offre d'autres possibilités : la sémantique *snapshot* peut être appliquée, mais XQuery ! rend aussi possible une spécification explicite de l'ordre d'évaluation, c'est-à-dire, le code peut décider de prendre en compte ses propres effets collatéraux. Par exemple, dans une fonction où une insertion est suivie d'une question sur le document, l'effet de l'insertion peut être pris en compte pour répondre à la question. Cela est mis en place par l'opérateur `snap{Expr}` qui évalue `Expr`, prend en compte ses réquisitions de mises à jour et les applique à la fin du *scope* de `snap` (sans attendre la fin de la fonction).

Nos primitives de mises à jour correspondent aux *update requests* de [GRS06] et, ainsi, nos listes de mises à jour (comme dans [GRS06]) représentent les mises à jour élémentaires dans le *scope* d'un `snap`. Néanmoins, XQuery ! prévoit trois sémantiques différentes pour l'application d'une liste de mise à jour (dans l'ordre, non déterministe et avec détection de conflit), alors que nous nous plaçons dans la sémantique avec détection de conflit. En effet, comme nous avons déjà mentionné, nous supposons l'existence d'un pré-traitement de la liste de mises à jour demandée par l'utilisateur : ce traitement peut proposer des modifications et des précisions de la part de l'utilisateur pour assurer l'obtention d'une liste de mises à jour non contradictoire (définition 3.2.2) qui est le seul type de liste de mises à jour traité par nos méthodes de validation.

La commutation des requêtes et mises à jour dans le langage XQuery ! est considérée dans [GRS07]. Les auteurs proposent une analyse statique qui fait l'inférence des chemins vers les nœuds atteints ou modifiés par une expression du langage. Cela correspond en fait à trouver un *projecteur* sur le document XML, notion que nous discutons dans le chapitre 6. Ensuite, ils proposent un théorème qui fournit une condition suffisante pour la commutation de deux expressions.

Dans le contexte de mises à jours, nous citons comme perspectives de recherche (voir chapitre 8) l'ajout des nouvelles primitives de mise à jour (par exemple, l'ajout d'un niveau dans la hiérarchie) ainsi que l'intégration de nos primitives de mises à jour à un langage de requêtes. Dans ce cadre, une possibilité intéressante (par son aspect *user friendly*) pourrait être l'utilisation des requêtes arbre (chapitre 6).

Chapitre 4

Mises à jour multiples et les contraintes de schéma : validation incrémentale et correction

Ce chapitre débute par la description de notre méthode de validation incrémentale par rapport aux contraintes de schéma. Cette méthode a été introduite dans [BH03] où les contraintes de schéma correspondaient à une grammaire d'arbres locale (LTG, voir section 2.2) et une seule mise à jour élémentaire était prise en compte. La méthode que nous expliquons ici est celle présentée[1] dans [BCH$^+$07, Che06] où sont considérés les langages réguliers d'arbres et les mises à jour multiples.

La validation incrémentale consiste à vérifier si l'arbre mis à jour respecte les contraintes de schéma, en testant seulement les parties de cet arbre concernées par les mises à jour dans une liste L. Notre validation *from scratch* correspond à un parcours à la SAX de l'arbre XML. Nous utilisons ce même parcours pour la validation incrémentale, seulement nous procédons à deux actions principales sur certaines positions spécifiques, à savoir : (1) quand une position de mise à jour p est atteinte, la mise à jour est prise en compte et (2) quand une position préfixe de p est atteinte (y compris la racine ϵ et la position de mise à jour p), une routine de validation locale est activée pour tester si le nœud en question reste valide suite à la mise à jour. À la fin du parcours, si la validité est préservée, l'arbre mis à jour devient le nouvel arbre, sinon, l'arbre original est conservé.

Au lieu de simplement refuser la liste de mises à jour demandées, un système de correction de mises à jour peut être mis en place. Ce système consiste en effet à corriger les documents XML en prenant en compte les mises à jour souhaitées. Un système de correction incrémentale des documents XML intégré à notre procédure de validation incrémentale (concernant les LTG et le $STTG$) est proposé dans la thèse de Ahmed Cheriat [Che06] que j'ai co-encadré. Ce travail (voir aussi [BCHS06b, BCHS06a]) est une extension, dans le domaine des arbres d'arité variable, des travaux sur la correction des mots et en particulier de la correction d'un mot par rapport à un langage, proposée dans [CSBH05][2]. Dans ce travaux nous avons compté avec l'importante collaboration d'Agata Savary qui avait déjà une très bonne connaissance du domaine de la correction de mots. Ceci étant, dans ce mémoire la partie correction sera traitée brièvement par une description rapide de notre procédure de correction intégrée à la validation incrémentale.

4.1 La méthode de validation incrémentale

Étant donné un arbre $T = (D, t)$ (définition 2.1.1) et une liste L de mises à jour (définition 3.2.3), la validation incrémentale consiste à vérifier si l'arbre mis à jour respecte les contraintes de schéma, en testant seulement les parties de T concernées par les mises à jour. L'arbre XML est visité dans le sens de lecture. Les mises à jour sont prises en compte à chaque position p dans L et les appels à la validation

[1] Rappel : Dans ce mémoire, nous ne traitons pas les attributs alors qu'ils sont pris en compte dans [BCH$^+$07, Che06].
[2] Ce travail est surtout le fruit des discussions entre Ahmed Cheriat et Agata Savary. Il ne sera pas traité ici.

locale sont fait à chaque position préfixe d'une position de mise à jour. L'exemple suivant illustre notre méthode de validation incrémentale.

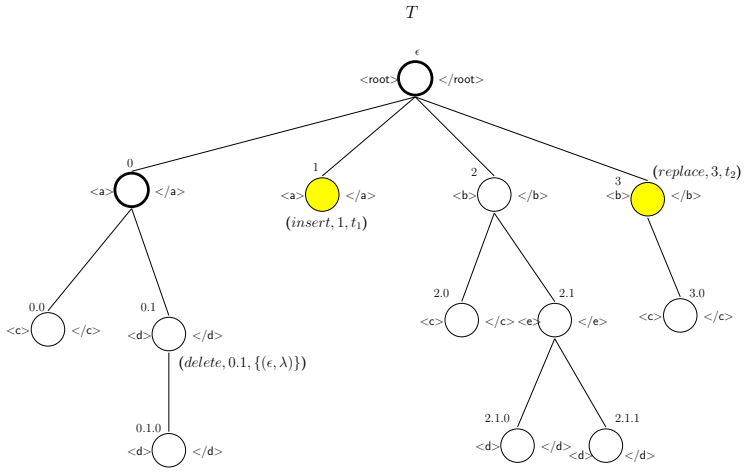


FIG. 4.1: Arbre XML et des opérations de mises à jour.

Exemple 4.1.1 La figure 4.1 illustre l'arbre XML T. Soit $L = \{(delete, 0.1, \{(\epsilon, \lambda)\}), (insert, 1, t_1), (replace, 3, t_2)\}$ une liste de mises à jour (définition 3.2.3) dont chaque mise à jour élémentaire est notée dans la figure 4.1. Soit $\mathcal{A}$ un automate d'arbre représentant les contraintes de schéma à vérifier.

Supposons que l'arbre original T (figure 4.1) est valide par rapport à $\mathcal{A}$ et que les deux arbres à insérer sont sont *localement valides* par rapport à $\mathcal{A}$. La procédure de validation incrémentale s'exécute en visitant tous les nœuds de l'arbre T mais en déclenchant une routine de validation locale seulement sur les nœuds en gras dans la figure 4.1 (c'est-à-dire, les nœuds qui sont concernés par les mises à jour dans L) :

- Quand la balise ouvrante <d> (à la position 0.1) est atteinte, l'opération $(delete, 0.1, \{(\epsilon, \lambda)\})$ est prise en considération et le sous-arbre enraciné à cette position est ignoré.
  Pour vérifier si l'opération de suppression $(delete, 0.1, \{(\epsilon, \lambda)\})$ est acceptée, nous devons considérer la règle de transition associée au nœud 0 (père de la position de suppression 0.1). Ce test est fait quand la balise fermante est trouvée (position 0). Notons que pour réaliser ce test, nous avons besoin de l'état associé à la position 0.0, mais pas de ses descendants. Les descendants du nœud 0.0 seront ignorés, même s'ils existent, car aucun d'eux n'est concerné par une opération de mise à jour.
- Quand la seconde balise ouvrante <a> (position 1) est trouvée, l'opération d'insertion $(insert, 1, t_1)$ est prise en compte et le nouveau sous-arbre t_1 (localement valide) est inséré à la position 1 (avant la balise <a>). En effet, si toutes les mises à jour sont acceptées, alors tous les frères droits de la position 1 seront décalés vers la droite (section 3.2)
  Ensuite, nous poursuivons la lecture des nœuds de positions 1 et 2 (par rapport à l'arbre d'origine). Notons qu'aucun descendant de la balise <b> (position 2) n'est vérifié, car aucun de ses descendants n'est concerné par une mise à jour dans L.
- L'opération de remplacement $(replace, 3, t_2)$ combine la suppression du sous-arbre de racine <b> à la position 3 et l'insertion du nouveau sous-arbre t_2 (localement valide) à la même position 3 (du document d'origine).
- La balise fermante </root> déclenche un test de validation qui prend en considération tous les fils de la racine *root*. Ce test consiste à vérifier si la règle de transition correspondante à *root* est respectée et un état final est trouvé à la racine. □

4.1.1 Algorithme de validation incrémentale

Dans la suite, nous avons fait le choix de rappeler l'algorithme de validation incrémentale proposé dans [BCH+07] puisque cette procédure guide toutes nos méthodes de validation. Cette section est donc une transcription des explications de l'algorithme que nous trouvons dans [BCH+07].

L'algorithme de validation incrémentale par rapport aux grammaires *RTG* est une extension de l'al-

gorithme de validation *from scratch* et leur structures de données de base sont les mêmes. Néanmoins, la validation incrémentale prend en compte les mises à jour et, dans certains cas (pour les grammaires RTG non $STTG$ et non LTG) utilise une structure auxiliaire pour stocker les résultats de la validation (réussie) précédente.

Deux structures de données sont nécessaires pour toutes nos formes de validation. Étant donné une position p de mise à jour, la première, appelée *états permis*, stocke les états *pouvant être associés* à p. La seconde contient, pour chaque position $p' \prec p$, les états qui *sont affectés* par le processus de validation aux fils de p'. Le processus de validation associe un ensemble d'états à un nœud p, si ce nœud respecte les contraintes correspondantes, imposées par le schéma. Ci-dessous, nous donnons les définitions de ces deux structures.

Définition 4.1.1 - États permis pour les fils d'une position p (*Permissible state for children of p*) : Soit $PSC(p)$ un ensemble d'états défini inductivement par :

$$\begin{aligned} PSC(\epsilon) &= \{q \mid \text{ il existe } \delta(a,E) = q_a \in \Delta \text{ telle que } t(\epsilon) = a \text{ et} \\ &\qquad q \text{ est un état qui apparaît dans } E \text{ et } q_a \in \mathcal{Q}_f\} \\ \\ PSC(p.i) &= \{q \mid \text{ il existe } \delta(a,E) = q_a \in \Delta \text{ telle que } t(p.i) = a \text{ et} \\ &\qquad q \text{ est un état qui apparaît dans } E \text{ et } q_a \in PSC(p)\} \end{aligned}$$ □

Pour chaque position $p.i$ (étiquetée a, fille de la position p), l'ensemble $PSC(p.i)$ contient les états *pouvant être* associés aux nœuds fils de $p.i$. Ces états sont ceux qui apparaissent dans l'expression régulière E, pour toute règle de transition associée à l'étiquette a (c'est-à-dire $\delta(a,E) = q \in \Delta$). Remarquer que nous considérons uniquement les règles de transition de la forme $\delta(a,E) = q \in \Delta$ pour lesquelles $q \in PSC(p)$. De cette manière, l'ensemble d'états permis pour p est déterminé en fonction de son contexte de filiation. Par exemple, considérons les deux règles de transition $\delta(a,E_1) = q_{a1} \in \Delta$ et $\delta(a,E_2) = q_{a2} \in \Delta$. Soient $struct_1$ et $struct_2$ deux éléments XML tels que $struct_1$ a un fils étiqueté a devant respecter la règle $\delta(a,E_1) = q_{a1}$ et $struct_2$ a un fils a devant respecter la règle $\delta(a,E_2) = q_{a2}$. Autrement dit, nous avons : pour définir les fils de $struct_1$, la règle $\delta(struct_1,E) = q$ où q_{a1} est dans E et pour définir les fils de $struct_2$, la règle $\delta(struct_2,E') = q'$ où q_{a2} est dans E'. Quand on cherche l'état associé à l'élément étiqueté a fils du nœud $struct_1$, l'état q_{a2} n'est pas considéré, car $PSC(p)$, où p est la position associée à $struct_1$, contient seulement q_{a1}. Ainsi, il n'y a que l'état q_{a1} qui peut être associé à l'élément a fils de $struct_1$. Remarquer que pour les schémas correspondants aux LTG et $STTG$, l'ensemble $PSC(p)$ contient un seul état pour chaque fils dû à l'unicité d'interprétation (section 2.2).

La définition suivante montre la façon dont on calcule une liste d'ensembles d'états, associée à chaque position p. Ces ensembles sont ceux associés par l'automate d'arbre aux fils de p. Cette liste est calculée en tenant compte des mises à jour à exécuter sur le document XML.

Définition 4.1.2 - États des fils d'une position p (*State attribution for children of p*) : Soit p une position d'un nœud dans un arbre XML T, la liste[3] $SAC(p)$ est composée de la concaténation des ensembles d'états associés par la vérification de schéma aux nœuds fils de la position p comme suit :

$SAC(p) = [\mathcal{Q}_{p.1}, \ldots, \mathcal{Q}_{p.n}]$ où, **pour tout** $1 \leq i \leq n$:

$$\mathcal{Q}_{p.i} = \begin{cases} \{\ \} & \text{Si } p.i \text{ est une position de suppression} \\ \Phi \cap PSC(p) & \text{Si } p.i \text{ est une position d'insertion ou de remplacement} \\ \{q \mid q \in \mathcal{Q}^{old}_{p.i}\} & \text{Si } p.i \text{ n'est pas un ascendant d'une position de mise à jour} \\ \{q \mid q \in PSC(p) & \text{(c'est-à-dire, il existe une règle } \delta(a,E) = q \in \Delta, \text{ telle que } t(p.i) = a), \text{ et} \\ \quad L(E) \cap L(SAC(p.i)) \neq \emptyset\}. & \text{Si } p.i \text{ est un ascendant d'une position de mise à jour} \end{cases}$$

L'ensemble Φ contient les états associés à la racine du sous-arbre étant inséré à la position p. Ainsi Φ est le résultat d'une validation locale réussie.

L'ensemble $\mathcal{Q}^{old}_{p.i}$ contient les états associés à $p.i$ pendant la validation précédente (avant les mises à jour). Si la validation est par rapport aux $STTG$ et LTG, nous définissons $\mathcal{Q}^{old}_{p.i} = \{q \mid q \in PSC(p), t(p.i) = a, \text{il existe une règle } \delta(a,E) = q \in \Delta\}$. Sinon, chaque ensemble $\mathcal{Q}_{p.i}$ est stocké dans une structure auxiliaire (et devient le $\mathcal{Q}^{old}_{p.i}$ pour la validation suivante).

[3] Dans [BCH+07, Che06] les attributs sont considérés et la liste contient comme premier élément l'ensemble $\mathcal{Q}_{att}$.

Le langage $L(SAC(p))$ est défini par l'expression régulière $(q_1^0 \mid q_1^1 \mid \ldots \mid q_1^{k_1}) \ldots (q_n^0 \mid q_n^1 \mid \ldots \mid q_n^{k_m})$ telle que $k_i = |\mathcal{Q}_{p.i}|$ et chaque $q_i^j \in \mathcal{Q}_{p.i}$ (où $1 \leq i \leq n$). □

La construction de chaque ensemble $\mathcal{Q}_{p.i}$ dans la liste $SAC(p)$ dépend de la situation de $p.i$ par rapport aux mises à jour. Quand $p.i$ est une position de mise à jour, l'ensemble $\mathcal{Q}_{p.i}$ prend en compte le type de la mise à jour. Si $p.i$ est un ascendant d'une position de mise à jour, alors un test de validation est nécessaire à la position $p.i$. Dans ce cas, $\mathcal{Q}_{p.i}$ est l'ensemble d'états associés à $p.i$ si la validation réussit à cette position. Si aucun des descendants de $p.i$ n'est concerné par une mise à jour, alors le contenu de $\mathcal{Q}_{p.i}$ peut être défini par $PSC(p)$ ou, si nous considérons la validation par rapport à une grammaire RTG qui n'est pas $STTG$ ni LTG, par une structure auxiliaire. En effet, dans ce dernier cas, la validation incrémentale nécessite d' une structure auxiliaire où, tout d'abord les résultats de la validation *from scratch* sont stockés et ensuite mis à jour par les validations réussies. Dans nos travaux nous n'avons pas considéré la mise en œuvre de cette structure auxiliaire, étant donné que nous tests ont été fait sur des grammaire $STTG$ et LTG.

Algorithme 4.1.1 - Incremental Validation of Multiple Updates [BCH+07] :

Input :
(*i*) *doc* : An XML document
(*ii*) $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$: A tree automaton
(*iii*) *UpdateTable* : A relation that contains updates to be performed on *doc*.
Each tuple in *UpdateTable* has the form $\langle pos, op, T_{pos}, \Phi \rangle$ where *pos* is an update position (considering the tree representation of *doc*), *op* is an update operation, T_{pos} is the tree to be inserted at position *pos* (when *op* is an insertion or a replace operation) and Φ is the set of states associated to the root of T_{pos} by the execution of $\mathcal{A}$ over T_{pos} (*i.e.*, the result of the local validation). All inserted subtrees are considered to be locally valid.
Output : If the XML document remains valid after all operations in *UpdateTable* the algorithm returns the boolean value *true*, otherwise *false*.

(1) **for** *each event v in the document*
(2) *skip*:= **false** ;
(3) **switch** *v* **do**
(4) **case** *start of element a at position p* :
(5) **if** *a* ≠"<root>"{
(6) **if** ∃ $u = (p, \mathsf{delete}, T_p, \Phi) \in UpdateTable$ **then** *skip*:= **true** ;
(7) **if** ∃ $u = (p, \mathsf{replace}, T_p, \Phi) \in UpdateTable$ **then** {
(8) Compute $\mathcal{Q}_p$ (Definition 4.1.2) ;
(9) **if** ($\mathcal{Q}_p = \emptyset$) **then** report "invalid" and halt ;
(10) $SAC(father(p)) = SAC(father(p))@\mathcal{Q}_p$;
//Append $\mathcal{Q}_p$ to $SAC(father(p))$
(11) *skip*:= **true** ; }
(12) **for each** $u = (p, \mathsf{insert}, T_p, \Phi) \in UpdateTable$ **do** {
(13) Compute $\mathcal{Q}_p$ (Definition 4.1.2) ;
(14) **if** ($\mathcal{Q}_p = \emptyset$) **then** report "invalid" and halt ;
(15) $SAC(father(p)) = SAC(father(p))@\mathcal{Q}_p$; }
(16) **if** $\nexists u' = (p', op', T', \Phi') \in UpdateTable$ *such that* $p \prec p'$ {
//If there is no update over a descendant of *p*
(17) Compute $\mathcal{Q}_p$ (Definition 4.1.2) ;
(18) $SAC(father(p)) = SAC(father(p))@\mathcal{Q}_p$;
(19) *skip*:= **true** ; }
(20) }
(21) **if** *a* ="<root>" **or** ¬*skip* **then** {
//If *p* is an ascendant of an update position
(22) Compute $PSC(p)$ (Definition 4.1.1) ;
(23) $SAC(p) = SAC(p)@\mathcal{Q}_{att}$;
//Starting the construction of the list $SAC(p)$ (with attributs, if they are considered) }
(24) **if** *skip* **then** $skipSubTree(doc, a, p)$; //Simple transversal of the subtree rooted at *p*. No validation.

(25) **case** *end of element a at position p* :

(26) **for each** $u = (p.i, \mathsf{insert}, T_{p.i}, \Phi) \in UpdateTable$ **and** $p.i$ is a frontier position **then** {

(27) Compute $\mathcal{Q}_{p.i}$ (Definition 4.1.2) ;

(28) **if** $(\mathcal{Q}_{p.i} = \emptyset)$ **then** report "invalid" and halt ;

(29) $SAC(p) = SAC(p)@\mathcal{Q}_{p.i}$; }

(30) Compute $\mathcal{Q}_p$ (Definition 4.1.2) ;

(31) **if** $(\mathcal{Q}_p = \emptyset)$ **then** report "invalid" and halt ;

(32) **if** $a \neq$"`</root>`" **then** $SAC(father(p)) = SAC(father(p))@\mathcal{Q}_p$;

(33) report "valid" □

L'algorithme 4.1.1 traite le document XML *doc* à la SAX [MB02]. En parcourant le document XML, l'algorithme 4.1.1 consulte *UpdateTable* pour décider quels sont les nœuds qui doivent être examinés et ceux qui vont être ignorés. En arrivant à une balise ouvrante représentant une position de mise à jour p, différentes actions sont effectuées selon le type de l'opération de mise à jour :

- **delete** - Le sous-arbre dont la racine est à la position p est ignoré. Si la mise à jour est acceptée, ce sous-arbre n'apparaîtra pas dans le résultat. Les descendants de p sont donc ignorés par le processus de validation (ligne 6).
- **replace** - Le sous-arbre dont la racine est à la position p est supprimé et un nouveau sous-arbre est inséré à la même position p. Autrement dit, l'arbre appelé T_p dans $UpdateTable$ est placé comme un sous-arbre de T à la position p. Si l'ensemble d'états $\mathcal{Q}_p$ est non vide, cela indique que T_p est localement valide à la position p. Dans ce cas $\mathcal{Q}_p$ doit être ajouté à la fin de la liste $SAC(father(p))$. La validation ignore le sous-arbre original à la position p en considérant son remplacement par T_p ($\mathcal{Q}_p$ est associé à la position p) (lignes 7-11).
- **insert** - Pour toute *insertion* à la position p, la procédure de validation est semblable à celle du remplacement (lignes 12-15), mais le sous-arbre (d'origine) enraciné à p *n'est pas ignoré*, car il apparaîtra dans le document mis à jour, à droite des sous-arbres insérés à la position p.

Quand on atteint une balise ouvrante correspondant à un nœud de position p étiqueté a n'ayant pas de descendants concernés par une mise à jour (lignes 16-19), ce nœud représente la racine d'un sous-arbre qui doit être ignoré. Dans le cas de la validation par rapport aux $STTG$ et aux LTG, l'ensemble $\{q \mid$ il existe une règle $\delta(a, E) = q \in \Delta$, telle que $q \in PSC(father(p))$ et $T(p) = a\}$ est ajouté à la fin de la liste $SAC(father(p))$ (définition 4.1.2). Le cas de la validation par rapport aux RTG (non $STTG$) nécessite d'une structure auxiliaire qui stocke le résultat de la validation précédente.

Nous utilisons *skipSubTree* (ligne 24) pour "ignorer les nœuds" jusqu'à ce qu'on atteigne une position importante pour la validation incrémentale. Notons qu'au moment où on atteint une telle position, *skipSubTree* change la valeur de la variable *skip*.

Quand on atteint une balise ouvrante <a> d'un nœud de position p ayant un ou plusieurs descendants concernés par des mises à jour, les structures $PSC(p)$ et $SAC(p)$ doivent être initialisées (lignes 21-23). L'ensemble $PSC(p)$ contient les états qui peuvent être associés aux différents nœuds fils de p. Pour ce faire, nous cherchons dans certaines règles de transition $\delta(a, E) = q \in \Delta$ associées à l'étiquette a, tous les états qui apparaissent dans l'expression régulière E. Les règles de transition $\delta(a, E) = q$ concernées sont celles qui ont à leur tête un état q appartenant à $PSC(father(p))$ (voir la définition 4.1.1).

Quand on atteint une balise fermante représentant un nœud de position p, nous vérifions, en premier lieu, si des opérations d'insertion à la frontière sont demandées (c'est-à-dire à une position $p.i \notin dom(T)$ telle que $p \in dom(T)$). Dans ce cas, les insertions sont traitées (lignes 26-29). En second lieu (lignes 30-32), nous vérifions si les fils de p respectent le schéma. En effet, atteindre une balise fermante correspondant à un nœud p (n'étant pas ignoré) signifie que toutes les mises à jour concernant les descendants de p ont été vérifiées : on peut donc vérifier le nœud p.

Les contraintes de schéma qui concernent le nœud courant p (étiqueté a) sont examinées en tenant compte de la liste $SAC(p)$ (c'est-à-dire $[\mathcal{Q}_{p.1}, \ldots, \mathcal{Q}_{p.n}]$) Notons que la liste $SAC(p)$ est complète au moment où on trouve la balise fermante . Rappelons que les ensembles $\mathcal{Q}_{p.1}, \ldots, \mathcal{Q}_{p.n}$ contiennent les états associés aux fils de l'élément p (dans l'ordre de leur apparition dans le document XML). En effet, le but est de trouver l'ensemble $\mathcal{Q}_p$ associé à un nœud p. Cet ensemble doit être ajouté à la fin de

la liste $SAC(father(p))$. Remarquer que c'est dans cette étape que l'ordre des états associés aux fils de p est vérifié. Le calcul de $\mathcal{Q}_p$ correspond au dernier cas de la définition 4.1.2.

Plus précisément, si nous considérons le langage $L(SAC(p))$ défini par l'expression régulière $(q_1^0 \mid q_1^1 \mid \ldots \mid q_1^{k_1}) \ldots (q_n^0 \mid q_n^1 \mid \ldots \mid q_n^{k_m})$ telle que $k_i = |\mathcal{Q}_{p.i}|$ et chaque $q_i^j \in \mathcal{Q}_{p.i}$ (où $1 \leq i \leq n$), alors l'ensemble d'états résultant $\mathcal{Q}_p$ se compose de tous les états q pour lesquels nous pouvons trouver, dans Δ, des règles de transition de la forme $\delta(a, E) = q$. Les états q concernés doivent respecter les propriétés suivantes[4] : (1) q est un état appartenant à $PSC(father(p))$; (2) $L(E) \cap L(SAC(p)) \neq \emptyset$.

Notons que l'algorithme 4.1.1 vérifie toutes les mises à jour qui concernent les descendants d'un nœud p avant d'examiner si ce dernier est valide.

La proposition ci-dessous renforce la discussion sur l'unicité de l'interprétation (section 2.2) pour les $STTG$ et les LTG et permet de simplifier l'algorithme 4.1.1 de telle manière que nous puissions représenter $SAC(p) = [\mathcal{Q}_{p.1}, \ldots, \mathcal{Q}_{p.n}]$ par un mot.

Proposition 4.1.1 *Étant donné un schéma définissant un langage d'arbres à type unique STTL et un document XML, pour chaque position p considérée dans l'algorithme 4.1.1, les ensembles d'états associés aux fils de p de type élément, et qui ne sont pas concernés par une suppression, sont des singletons, c'est-à-dire si* $SAC(p) = [\mathcal{Q}_{p.1}, \ldots, \mathcal{Q}_{p.n}]$ *alors* $| \mathcal{Q}_{p.i} | = 1$ *pour tout* $1 \leq i \leq n$. □

Preuve : Voir [BCH+07]

Pendant le test de validation des mises à jour sur un document XML T, un nouveau document XML mis à jour T' peut être construit (comme copie modifiée du document original T). Si l'examen de la validation incrémentale est réussi, l'ancien document T est remplacé par le nouveau T' et celui-ci est considéré comme le document en cours. Si par contre le test de la validation incrémentale échoue, alors aucune mise à jour ne sera exécutée et le document original T est gardé sans aucune modification.

Complexité et résultats expérimentaux

Pour calculer la complexité de notre méthode, nous tenons compte du fait que le coût "des nœuds ignorés" (*Skip*) est négligeable quand on le compare avec le coût d'un *test de validation*. D'après l'algorithme 4.1.1, pour $T = (D, t)$, un test de validation est exécuté pour chaque nœud $p \in D$ ascendant d'une position de mise à jour. Soit E une expression régulière définissant la structure des fils de p. Soit n le nombre maximum de fils d'un nœud quelconque de T. Dans le cas des schémas correspondant à une LTG ou une $STTG$, chaque test de validation consiste à vérifier si un mot w appartient au langage $L(E)$, tel que w représente la concaténation des états associés aux fils du nœud p. Dans ce cas, un *test de validation* est exécuté au pire en $O(n)$. Dans le cas des schémas correspondant à une RTG, un *test de validation* est exécuté au pire en $O(n^2)$. Cela s'explique puisque chaque étape de validation consiste à vérifier s'il existe un mot $w = q_i \ldots q_n \in L(E)$, où $q_i \in \mathcal{Q}_{p.i}, \ldots, q_n \in \mathcal{Q}_{p.n}$. Pour cela il faut vérifier si $L(E_{aux}) \cap L(E) \neq \emptyset$, où $E_{aux} = (q_1^0 \mid q_1^1 \mid \ldots \mid q_1^{k_1}) \ldots (q_n^0 \mid q_n^1 \mid \ldots \mid q_n^{k_n})$. Cette vérification est faite en calculant l'intersection des deux automates $M_{E_{aux}}$ et M_E. La solution du problème est connue pour être $O(n^2)$ [HMU01, KLV00].

Soit m le nombre de mises à jour à exécuter sur T dont la profondeur est h. Étant donné une position de mise à jour p, au pire, un test de validation doit être fait pour chaque nœud appartenant au chemin entre p et la racine de T. Pour une analyse du plus mauvais cas, nous pouvons considérer également que : (1) chacune des m opérations de mises à jour est exécutée sur une feuille de T ; (2) tous les chemins entre les positions p de mises à jour et la racine sont disjoints. Dans ce cas, notre algorithme est en $O(m.n.h)$ (dans le cas des LTG et $STTG$) et en $O(m.n^2.h)$ (dans le cas des RTG). Si $T = (D, t)$ est un arbre *équilibré*, de profondeur $h = \log_n |t|$, où $|t|$ est le nombre de positions de D, notre algorithme est $O(m.n. \log_n |t|)$ (pour une $STTG$ ou LTG) et $O(m.n^2. \log_n |t|)$ (pour une RTG).

Dans la pratique, les mises à jour peuvent se produire à n'importe quel niveau de l'arbre (donc la contrainte imposée par h est rarement atteinte). De plus, dans le cas des mises à jour multiples, plusieurs chemins entre les positions de mises à jour et la racine vont se croiser (avant d'atteindre la racine). Dans ce cas, un seul test de validation est effectué au niveau du nœud où ces chemins se croisent.

Les résultats expérimentaux présentés dans [BCH+07, Che06] montrent que notre algorithme est très

[4] Toujours dans le cadre où les attributs ne sont pas pris en compte. Voir [BCH+07].

efficace en pratique. Nous avons fait une comparaison entre notre méthode de validation incrémentale (algorithme 4.1.1) avec Xerces [Xer] qui fait seulement la validation *from scratch*. D'ailleurs, quand on considère seulement la revalidation *from scratch* les tests ont montré que Xerces [Xer] est supérieur ; résultat normal puisqu'il s'agit de comparer un prototype avec un produit commercial. En revanche, la comparaison entre notre méthode de validation incrémentale (algorithme 4.1.1) et Xerces (pourtant implémentée efficacement) a donnée (pour des documents de grande taille[5]) un considérable avantage à notre méthode (voir [BCH+07] pour les détails des tests).

4.1.2 Travaux liés

La validation incrémentale est très utile lors du passage à une grande échelle car elle rend possible la vérification partielle des documents en contribuant à une réduction considérable du temps de validation.

Des algorithmes de validation par rapport aux schéma sont proposés dans [MLM01], mais les versions incrémentales ne sont pas considérées. Les travaux dans [BPV04, PV03] sont parmi les plus cités dans ce contexte et, comme nous, traitent de la validation incrémentale pour des arbres appartenant aux langages LTL, $STTL$ et RTL. La méthode proposée voit les documents comme des arbres binaires et utilise un automate ascendant. Deux algorithmes de validation incrémentale sont proposés pour vérifier m mises à jour sur un document T. Le premier traite les schémas du type LTG et $STTG$ et présente une complexité en temps de $O(m.log|T|)$; alors que le deuxième considère les RTG et présente une complexité en temps de $O(m.log^2|T|)$. Dans les deux versions, un structure auxiliaire de taille $O(|T|)$ est nécessaire.

Notre approche considère les mises à jour multiples et la validation incrémentale sur des arbres d'arité variable. Même si tous les nœuds d'un arbre sont visités, seulement certains nœuds activent les routines de validation. Ainsi, en termes de complexité notre approche est similaire à celle de [BPV04, PV03]. Puisque nous utilisons les arbres d'arité variable et non les arbres binaires, chaque étape de validation de notre méthode est plus chère, mais, en contre partie, notre arbre XML est beaucoup moins haut. Nous rappelons les principaux aspects qui différencient notre approche de [PV03, BPV04] :

1. Nos opérations de mises à jour peuvent être appliquées sur n'importe quel nœud de l'arbre et non seulement sur les feuilles.
2. Dans [PV03], le test d'appartenance d'un mot à un langage est fait de façon incrémentale, en stockant des informations concernant le document dans une structure auxiliaire. Cette optimisation peut être intégrée dans notre algorithme, mais elle semble intéressante uniquement pour des documents dont la plupart des nœuds ont un *fan out* élevé.
3. Pour effectuer une validation incrémentale par rapport aux $STTG$ et LTG dans notre approche, nous n'avons pas besoin de structures auxiliaires pour stocker les informations concernant des résultats de validations précédentes.
4. La vérification des contraintes d'intégrité peut être intégrée dans notre algorithme.

4.2 La correction incrémentale comme solution pour la violation des contraintes

Pendant l'exécution de notre méthode de validation incrémentale, si l'une des contraintes de schéma n'est pas respectée, nous pouvons déduire que le document est invalide ; la validation est interrompue et la mise à jour refusée. Néanmoins, une autre démarche consiste à effectuer des corrections sur les parties où se trouvent les erreurs et ensuite continuer la validation. Ce thème a été le centre de la thèse d'Ahmed Cheriat qui détaille l'approche que nous avons proposée dans [BCHS06b, BCHS06a]. Le problème de la correction intégrée dans la validation peut être défini comme suit :

> Soient un schéma Υ, un arbre XML T valide par rapport à Υ. Nous souhaitons utiliser l'algorithme 4.1.1 de validation incrémentale pour effectuer une mise à jour multiple (définition 3.2.3) sur T. Si l'algorithme 4.1.1 échoue dans une position p de l'arbre, comment corriger

[5]Pour des petits documents, le nombre de mises à jour représente un changement considérable par rapport à leurs tailles. Dans ce cas, le coût de la validation incrémentale est proche de celui de notre validation entière et ainsi supérieur à celui du produit commercial Xerces.

le sous arbre dont la racine est p avec un coût minimum et dans la limite d'un seuil d'erreur donné th, pour pouvoir continuer la validation incrémentale ?

Si la validation incrémentale échoue à la position p, une méthode de correction sera appelée. Le but de cette méthode est de corriger le sous-arbre invalide T_p^{sub} enraciné à p. Pour ce faire, nous considérons que l'étiquette l de la racine de T_p^{sub} est correcte et nous examinons la correction de ses descendants. Pour construire un nouveau sous-arbre valide $T_p^{sub'}$ de racine l, l'algorithme de correction peut proposer des modifications sur les étiquettes des fils de p. Puis, ces modifications peuvent engendrer des changements sur les petits-fils de p et ainsi de suite, jusqu'à ce qu'on atteigne les feuilles de T_p^{sub}. L'exemple suivant illustre notre méthode de correction.

Exemple 4.2.1 Considérons le document XML de la figure 4.2, qui est valide par rapport à un schéma de contraintes décrit par l'automate d'arbre $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, tel que $Q_f = \{root\}$ (supposons $Q = \Sigma$ pour simplifier la notation). L'ensemble Δ contient l'ensemble des règles de transition ci-dessous :

(1) $\delta(root, (a^*.b^*)) = root$
(2) $\delta(a, ((c.d)^* \mid m^*)) = a$
(3) $\delta(b, (c.e^*)) = b$
(4) $\delta(c, (g^*.f?)) = c$
(5) $\delta(d, (d^*)) = d$
(6) $\delta(e, (d^*)) = e$
(7) $\delta(m, g^+) = m$
(8) $\delta(g, \epsilon) = g$
(9) $\delta(f, \epsilon) = f$

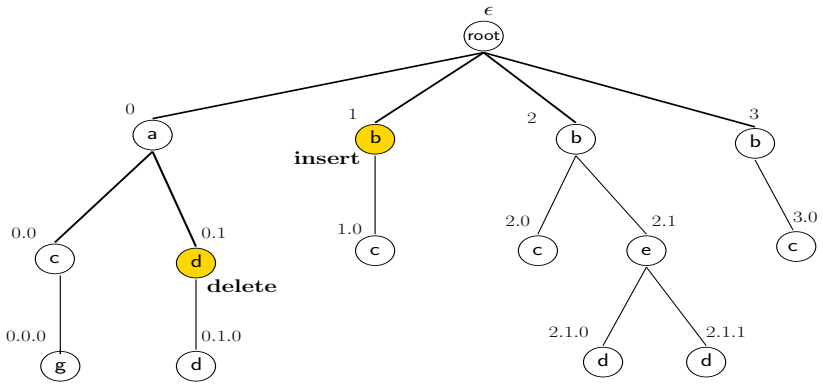


FIG. 4.2: Arbre XML et des opérations de mises à jour.

Considérons une séquence de mises à jour contenant la suppression du sous-arbre enraciné à la position 0.1 et l'insertion du sous-arbre localement valide $\tau_1 = \{(\epsilon, a)(0, c)(1, d)\}$ à la position 1. La validation incrémentale échoue à la position 0, car le mot d'état $w = c$ (résultant de la suppression du sous-arbre 0.1) ne respecte pas l'expression régulière $E_a = (c.d)^* \mid m^*$ de la règle de transition associée à a (c'est-à-dire $w \notin L(E_a)$).

La première correction possible est obtenue par la réinsertion du nœud d à la position 0.1 avec le coût 1. La deuxième correction consiste à supprimer le sous-arbre enraciné à la position 0.0. Le coût de cette opération est égal à 2, car elle correspond à la suppression des nœuds 0.0 et 0.0.0. La troisième correction peut être établie par le renommage de la racine du sous-arbre enraciné à 0.0 par m. En effet, comme m doit respecter l'expression régulière $E_m = g^+$, il suffit dans ce cas de renommer le nœud 0.0 par m et de garder l'étiquette du nœud 0.0.0. Le coût de cette troisième correction est également 1. Si nous considérons que le seuil d'erreur th est égal à 2, on peut ainsi proposer à l'utilisateur chacune de ces trois solutions comme correction possible. □

Par analogie avec les chaînes de caractères, la correction d'un arbre T n'appartenant pas à un langage d'arbre $\mathcal{L}$ consiste à déterminer les modifications minimales à effectuer sur T, pour avoir un arbre appartenant à $\mathcal{L}$. Trouver la différence minimale entre deux arbres, ou corriger un arbre par rapport à un autre, consiste à (i) déterminer les opérations d'édition pouvant être appliquées sur un arbre T pour trouver T' et (ii) définir la distance d'édition entre T et T' à partir de ces opérations.

Dans nos travaux, nous considérons la validation incrémentale par rapport aux LTG et $STTG$ et nous proposons une méthode de correction qui est activée chaque fois que la validation échoue. À la fin de la validation, les corrections produites par chaque appel à la méthode de correction sont combinées et

plusieurs versions de corrections sont proposées à l'utilisateur.

Dans la suite, les définitions concernant la correction des documents XML ainsi que notre routine de correction ne sont pas présentées dans tous les détails. Notre option est de simplement introduire les concepts de base nécessaires à la compréhension de la méthode pour ensuite faire une description de notre approche de correction. Un exemple illustre comment le calcul de la correction est fait pas à pas. Nous référençons [BCHS06a, BCHS06b, Che06] pour les détails de la méthode.

4.2.1 Préliminaires

Dans cette section, nous introduisons des notations et des concepts nécessaires pour comprendre notre méthode de correction.

Notation sur les arbres

Certaines notations spécifiques seront utilisées pour la présentation de notre méthode de correction

- Dans la définition 2.1.2 nous avons introduit le concept de sous-arbre $T_p^{sub} = (D_p^{sub}, t_p^{sub})$ d'un arbre $T = (D, t)$. Nous notons $T_p^{tree} = (D_p^{tree}, t_p^{tree})$ l'arbre obtenu du sous-arbre T_p^{sub} tel que $D_p^{tree} = \{s \mid p.s \in D \text{ and } s \in \mathbb{N}^*\}$ et pour chaque $s \in D_p^{tree}$ nous avons $t_p^{tree}(s) = t(p.s)$.
- Un **arbre partiel** $T\langle i\rangle = (D\langle i\rangle, t\langle i\rangle)$ est composé de la racine de T et des sous-arbres $T_0^{sub}, \ldots T_i^{sub}$ (figure 4.3(a)).
- Soit $\mathcal{L}$ un langage d'arbre défini par un automate d'arbre $\mathcal{A}$. Définir $\mathcal{L}_l \subseteq \mathcal{L}$ comme le langage d'arbre qui contient tous les arbres de $\mathcal{L}$ qui ont une racine étiquetée l. Ce langage est défini par une automate d'arbre similaire à $\mathcal{A}$ mais où l'ensemble d'états finaux[6] est $Q_f = \{l\}$.

Les opérations d'éditions et leurs coûts

L'idée de base de notre méthode de correction est la comparaison d'un arbre non valide avec un arbre valide. Le but de cette comparaison est savoir comment et à quel coût il est possible de transformer l'arbre non valide dans l'arbre valide en utilisant nos opérations de mise à jour (définition 3.2.1). Or, comme nos opérations de mises à jour sont des opérations sur des arbres, l'estimation de leur coût doit prendre en compte la taille des arbres. Nous avons donc considéré des *opérations d'édition* sur les nœuds des arbres ; le but étant de les prendre comme des primitives sur lesquelles les opérations de mise à jour de la définition 3.2.1 pourraient être décomposées. Ainsi, dans [BCHS06b, BCHS06a, Che06] trois *opérations d'édition* sur un arbre $T = (D, t)$ ont été introduites, à savoir : (i) *relabel* qui change l'étiquette associée au nœud à la position $p \in D$; (ii) *remove* qui permet la suppression d'un nœud feuille et (iii) *add* qui permet l'ajout d'un nœud feuille à une position $p \in D$ ou $p \in fr^{ins}(T)$ telle que $p \neq \epsilon$. Chaque opération de mise à jour correspond à une séquence (composition) d'opérations d'édition :

$(insert, p, \Gamma)$: L'insertion d'un arbre Γ à la position p dans un arbre T correspond à l'ajout des nœuds de Γ, dans T, un par un. Pour minimiser le nombre de décalages ($ShiftRightPos$), les nœuds de Γ sont insérés dans T, en commençant par la racine de Γ jusqu'à ses feuilles et de la gauche vers la droite.

$(delete, p, \Gamma)$: La suppression d'un sous-arbre enraciné à la position p dans un arbre T correspond à la suppression de tous les descendants de p un par un, dans l'ordre suivant : des feuilles du sous-arbre T_p^{sub} vers la racine p et de la droite vers la gauche.

$(replace, p, \Gamma)$: Le remplacement d'un sous-arbre, à la position p dans un arbre T, par un autre arbre Γ est défini comme une séquence d'opérations élémentaires (*add*, *remove* et *relabel*) permettant de transformer T_p^{sub} en Γ. Cette séquence d'opérations d'édition est déterminée de façon telle qu'on doit effectuer le minimum de changements possible sur T_p^{sub} pour avoir Γ.

Nous considérons que le coût associé à chaque opération d'édition (ed), dénoté par $cost(ed)$, est égal à 1. Étant donné l'opération de mise à jour $T \xrightarrow{upd} T'$ correspondant à la séquence $T = T_0 \xrightarrow{ed_1} T_1 \xrightarrow{ed_2} T_2 \ldots \xrightarrow{ed_n} T_n = T'$, le coût de *upd*, noté $Cost(upd)$, est défini par $Cost(upd) = \Sigma_{i=1}^{n}(cost(ed_i))$. La notion de mise à jour multiple (définition 3.2.3) généralise ce concept et son coût est définie dans la suite.

[6] Ici, pour ne pas alourdir la notation, l est l'état associé au label l.

Définition 4.2.1 - Coût d'une mises à jour multiple : Soit $L = upd_1 \dots upd_k$ une séquence (liste) d'opérations de mises à jour (comme dans la définition 3.2.3) . Le coût associé à S est défini par $Cost(L) = \Sigma_{i=1}^{k} Cost(upd_i)$. Si $k = 0$ alors $Cost(L) = 0$. □

Étant donnés deux arbres T et T', plusieurs séquences de mises à jour $L_1, \dots, L_n$ peuvent exister telles que $T \xrightarrow{L_i} T'$ où $1 \leq i \leq n$. La distance d'édition est le minimum des coûts de ces séquences. Formellement, la distance d'édition entre deux arbres est définie comme suit.

Définition 4.2.2 - Distance d'édition entre arbres [BCHS06b] : Soit T et T' deux arbres. Soit S l'ensemble de toutes les séquences de mises à jour L_i permettant, chacune d'elles, de transformer T en T'. La distance d'édition entre T et T' est définie par $dist(T, T') = min_{L_i \in S}\{Cost(L_i)\}$. La distance d'édition entre un arbre T et un langage d'arbre $\mathcal{L}$ est définie par $DIST(T, \mathcal{L}) = min_{T' \in \mathcal{L}}\{dist(T, T')\}$ □

4.2.2 Correction d'un arbre invalide

Dans [BCHS06b, BCHS06a] étant donné un arbre XML T pour lequel le processus de validation échoue à la position p (c'est-à-dire $T_p^{tree} \notin \mathcal{L}_l$), nous proposons une méthode permettant de corriger T_p^{sub}. Dans cette méthode nous partons du principe que l'étiquette l de la racine p de T_p^{sub} est correcte et nous analysons les possibles modifications sur les descendants de p. À partir du langage $\mathcal{L}_l$ et de l'arbre $T_p^{tree} \notin \mathcal{L}_l$, notre algorithme détermine les nouveaux arbres $T_p^{tree'} \in \mathcal{L}_l$, tels que $dist(T_p^{tree}, T_p^{tree'})$ est minimale. Nous rappelons que chaque nouvel arbre $T_p^{tree'}$ peut être représenté comme un sous-arbre $T_p^{sub'}$.

Exemple 4.2.2 Dans l'exemple 4.2.1, notre procédure de validation échoue à la position 0 de l'arbre XML et notre méthode de correction peut être exécutée. En effet, comme $t_0^{tree} = \{(\epsilon, a), (0, c), (0.0, g)\}$, donc $t_0^{tree} \notin \mathcal{L}_a$. Pour corriger le sous-arbre invalide T_0^{sub} de racine a, cette méthode considère le langage d'arbres $\mathcal{L}_a$ défini par l'automate d'arbre dont les règles de transition sont celles montrées dans l'exemple 4.2.1 mais avec $Q_f = \{a\}$. □

REMARQUE : Dans la suite de cette section et afin de simplifier la notation, nous utilisons $T = (D, t)$ comme notation pour l'arbre T_p^{tree} et $T' = (D', t')$ pour noter l'arbre $T_p^{tree'}$.

Notre algorithme est une extension des algorithmes proposés dans [Sel77] et dans [Ofl96] : il utilise une matrice de distance d'édition H_DIST. Chaque élément $H\_DIST[i, j]$ contient la distance d'édition entre deux arbres *partiels* $T\langle i \rangle$ et $T'\langle j \rangle$.

La matrice H_DIST est calculée colonne par colonne et chaque nouvel élément $H\_DIST[i, j]$ est déduit à partir de ses trois voisins $H\_DIST[i-1, j-1]$, $H\_DIST[i-1, j]$ et $H\_DIST[i, j-1]$ qui sont calculés auparavant (voir figure 4.3(b)). En effet, chaque élément $H\_DIST[i, j]$ contient le couple (*coût*, *séquence d'opérations de mises à jour*). Ce couple est obtenu à partir de ses trois voisins et des mises à jour qui sont nécessaires pour :

- supprimer le sous-arbre T_i^{sub} (figure 4.3(b), arc (3)),
- insérer un nouveau sous-arbre $T_j^{'sub}$ (figure 4.3(b), arc (1)) et
- remplacer le sous-arbre T_i^{sub} par un nouveau sous-arbre $T_j^{'sub}$ (figure 4.3(b), arc (2)).

Le calcul de chaque élément $H\_DIST[i, j]$ suit le raisonnement "horizontal-vertical" illustré dans la figure 4.4. Notons que $T = (D, t)$ est l'arbre à corriger et $T' = (D', t')$ représente l'arbre que nous allons construire, tel que les fils de la racine de T' doivent respecter l'expression régulière $E_{t(\epsilon)}$. Considérons le mot $w \notin L(E_{t(\epsilon)})$ composé de la concaténation des états associés aux nœuds fils de la racine de T.

Sur le *plan "horizontal"*, notre algorithme de correction s'inspire de notre proposition dans [CSBH05] pour la correction d'un mot par rapport à un langage des mots. Nous cherchons tous les mots $w' \in L(E_{t(\epsilon)})$ les plus proches possibles de w en utilisant le seuil d'erreur et l'automate à états finis $A_{t(\epsilon)}$ associé au langage $L(E_{t(\epsilon)})$. Pour toute transition de $A_{t(\epsilon)}$ utilisée, une nouvelle colonne est ajoutée à la matrice de distance d'édition. Si un état final de $A_{t(\epsilon)}$ est atteint et si l'élément le plus bas de la dernière colonne n'excède pas un seuil donné, alors le mot courant w' est un candidat de correction valide. Toutefois, si aucun élément de la colonne courante n'excède le seuil, nous pouvons continuer à chercher d'autres candidats en utilisant la transition suivante de $A_{t(\epsilon)}$ (si elle existe). Dans le cas où un élément de la colonne courante dépasse le seuil, nous devons faire un retour en arrière en supprimant la colonne courante et en utilisant une transition différente.

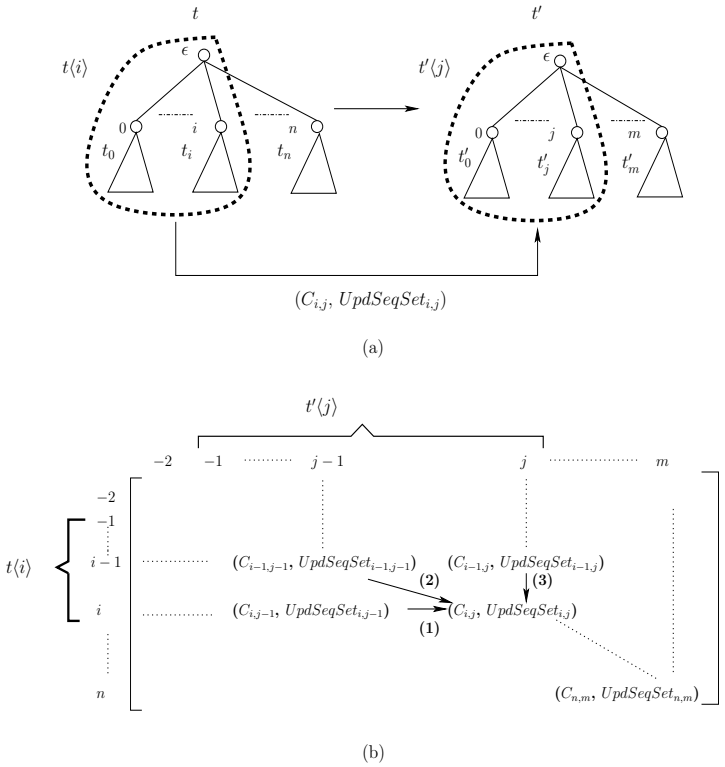


FIG. 4.3: (a) Deux arbres (partiels) : Illustration de $t\langle i\rangle$ et $t'\langle j\rangle$. (b) La matrice H_DIST associée aux arbres T et T' et le calcul de $H\_DIST[i,j]$.

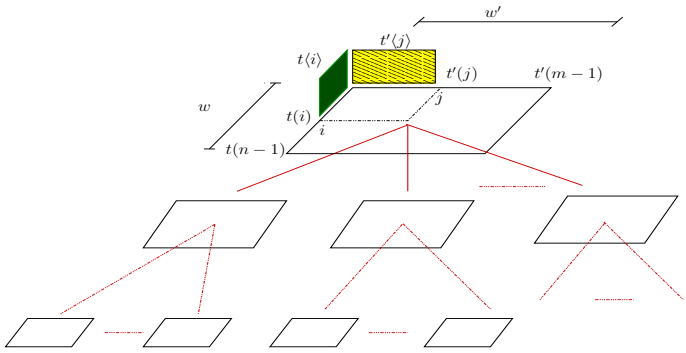


FIG. 4.4: Schéma descriptif de notre méthode de calcul de la matrice de distance d'édition entre un arbre $t \notin \mathcal{L}$ et un arbre $t' \in \mathcal{L}$.

Cependant, dans le contexte des arbres, chaque "caractère" dans w représente lui-même l'étiquette de la racine d'un sous-arbre. Afin d'obtenir w', nous devons également travailler dans la direction "verticale". En effet, chaque $w'[j]$ doit être à la "racine" d'un sous-arbre $T_j^{'sub}$, c'est-à-dire, à la racine de $T_j^{'tree} = (D_j^{'tree}, t_j^{'tree})$ qui doit appartenir à $\mathcal{L}_{t'(j)}$, où $t_j^{'tree}(\epsilon) = w'[j] = t'(j)$.

Tandis que sur le plan "horizontal" (figure 4.4) nous traitons une matrice qui associe un mot w à un langage régulier de mots, sur le *plan "vertical"*, nous traitons un arbre T_i^{sub}. En effet, pour corriger T_i^{sub}, il n'est pas suffisant de calculer une matrice de distance pour les fils de la racine de T_i^{sub} mais il peut être nécessaire de le refaire pour tous ses descendants, comme le montre la figure 4.4.

La définition de H_DIST n'est pas présentée ici (voir [BCHS06b, BCHS06a, Che06]), mais nous rappelons un exemple utilisé dans nos publications pour expliquer la routine de correction.

Exemple 4.2.3 Nous avons vu dans l'exemple 4.2.2, que le test de validation échoue à la position 0 et la procédure de correction est appelée pour corriger l'arbre T_0^{tree} en considérant le langage d'arbres $\mathcal{L}_a$ et le seuil d'erreur $th = 2$.

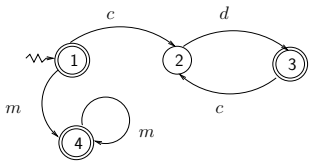


FIG. 4.5: L'automate à états finis A_E associé à l'expression régulière $E_a = (cd)^*|m^*$.

L'automate à états finis A_E, correspondant à l'expression régulière $E_a = (c.d)^* \mid m^*$, est montré dans la figure 4.5. Dans ce qui suit, nous décrivons comment notre méthode de correction détermine les candidats de correction de l'arbre T_0^{tree}.

Étape 1 : Soit $T = (D, t)$ l'arbre que nous considérons dans la procédure de correction où $t = \{(\epsilon, a), (0, c), (0.0, g)\}$. Dans l'initialisation de H_DIST_1, chaque élément $H\_DIST[i, -2]$ de la première colonne, ainsi que chaque élément $H\_DIST[-2, j]$ de la première ligne, reçoit un coût très grand (∞). La colonne -2 correspond à une borne maximale permettant de limiter le calcul de la matrice. Notons que la ligne -1 représente le nœud racine étiqueté a dont les fils sont représentés par les lignes $i \geq 0$ de H_DIST_1. Dans A_E, l'état 1 qui représente à la fois l'état initial et final devient l'état courant.

La Matrice de distance d'édition H_DIST_1 à l'initialisation :

			a
		-2	-1
	-2	$(\infty, \top)$	$(\infty, \top)$
a	-1	$(\infty, \top)$	$(0, \{[]\})$
c	0	$(\infty, \top)$	

Comme nous considérons que l'étiquette de la racine de t ne change pas (elle reste a), l'élément H_DIST_1 $[-1, -1] = (0, \{[]\})$. Pour calculer $H\_DIST_1[0, -1]$, nous considérons ses trois voisins, H_DIST_1 $[-1, -1]$, H_DIST_1 $[0, -2]$ et H_DIST_1 $[-1, -2]$, qui sont déjà calculés. Dans notre cas, les deux voisins H_DIST_1 $[0, -2]$ et $H\_DIST_1[-1, -2]$ sont écartés, car leur coûts respectifs (∞ pour chacun des deux) excèdent le seuil. Pour calculer $H\_DIST_1[0, -1]$ à partir de $H\_DIST_1[-1, -1]$, nous devons considérer la suppression du sous-arbre enraciné à la position 0 de T_0^{tree}.

Cette suppression, est traitée en effectuant un appel récursif à la routine de correction (c'est-à-dire, en commençant l'étape 2) avec les paramètres suivants : (i) l'arbre obtenu à partir du sous-arbre enraciné à la position 0 de T_0^{tree} et (ii) le langage d'arbres $\mathcal{L}_\lambda$ qui ne contient que l'arbre vide.

Étape 2 : Nous allons vers le bas dans la direction verticale (figure 4.4) et nous commençons la construction de la nouvelle matrice H_DIST_2. Pour l'arbre $T = (D, t)$ que nous considérons dans ce cas nous avons $t = \{(\epsilon, c), (0, g)\}$. La matrice H_DIST_2 est initialisée :

Initialisation de la matrice H_DIST_2 :

			λ
		-2	-1
	-2	$(\infty, \top)$	$(\infty, \top)$
c	-1	$(\infty, \top)$	$(1, \{(remove, \epsilon, \lambda)\})$
g	0	$(\infty, \top)$	

Remarquons que $H\_DIST_2[-1, -1] = (1, \{(remove, \epsilon, \lambda)\})$, car il est calculé en considérant la suppression de $t(\epsilon) = c$. Comme dans l'étape 1, l'élément $H\_DIST_2[0, -1]$ est calculé à partir de $H\_DIST_2[-1, -1]$, en considérant la suppression de $\{(0, g)\}$ de t. Cette suppression est traitée par l'appel récursif qui commence l'étape 3 ci-dessous.

Étape 3 : Nous allons encore vers le bas dans la direction verticale (figure 4.4) et nous commençons la construction de la nouvelle matrice H_DIST_3. L'arbre T que nous considérons dans ce cas est $t = \{(\epsilon, g)\}$. La matrice H_DIST_3 est initialisée et comme T n'a qu'un nœud unique, nous obtenons :

Matrice H_DIST_3 :

			λ
		-2	-1
	-2	$(\infty, \top)$	$(\infty, \top)$
g	-1	$(\infty, \top)$	$(1, \{(remove, \epsilon, \lambda)\})$

Retour à l'étape 2 : Le résultat de l'étape 3 est retourné (par le retour de l'appel récursif) à H_DIST_2. Notons que, par rapport à l'arbre T que nous avons considéré à l'étape 2, le résultat de l'étape 3 correspond à la suppression du nœud de position 0. La suppression de la racine de T (de l'étape 2) est également considérée. Ainsi, H_DIST_2 devient :

Matrice H_DIST_2 :

			λ
		-2	-1
	-2	$(\infty, \top)$	$(\infty, \top)$
c	-1	$(\infty, \top)$	$(1, \{(remove, \epsilon, \lambda)\})$
g	0	$(\infty, \top)$	$(2, \{(remove, 0, \lambda)(remove, \epsilon, \lambda)\})$

Le dernier élément de la dernière ligne de la matrice H_DIST_2 est retourné, à son tour, à l'étape 1.

Retour à l'étape 1 : Le résultat de l'étape 2 est retourné, par le retour de l'appel récursif, à H_DIST_1. Ce résultat correspond, par rapport à l'arbre T que nous avons considéré à l'étape 1, à la suppression du sous-arbre enraciné à la position 0.0 . Ainsi, nous avons :

Matrice H_DIST_1 :

			a
		-2	-1
	-2	$(\infty, \top)$	$(\infty, \top)$
a	-1	$(\infty, \top)$	$(0, \{[]\})$
c	0	$(\infty, \top)$	$(2, \{(remove, 0.0.0, \lambda)(remove, 0.0, \lambda)\})$

L'état actuel de A_E (qui est l'état 1) est aussi un état final et le dernier élément de la dernière ligne de H_DIST_1 n'a pas dépassé le seuil $th = 2$. Ainsi, selon [Ofl96, Sel77], l'élément $H\_DIST_1[0, -1]$ contient un candidat de correction valide. La séquence permettant de générer ce candidat à partir de T où $t = \{(\epsilon, a), (0, c), (0.0, g)\}$ et la suppression du sous-arbre de racine 0. Cela correspond à la suppression des nœuds 0.0 et 0.0.0 dans l'arbre de la figure 4.2. Notons que, dans ce cas, le mot w' de la figure 4.4 est le mot vide qui appartient à $L(E_a)$, pour $E_a = (c.d)^* \mid m^*$.

Pour trouver plus de solutions respectant le seuil 2, nous considérons d'autres mots $w' \in L(E_a)$. Pour ce faire, nous pouvons examiner d'autres transitions sortantes de l'état courant 1 de l'automate A_E. Ainsi, nous pouvons, par exemple, considérer la transition étiquetée m aboutissant à l'état 4 qui permettra d'ajouter une nouvelle colonne à H_DIST_1. La routine de correction retourne la matrice H_DIST (décrite ci-dessous), telle que la solution proposée dans ce cas consiste à renommer le nœud de position 0.0, dans l'arbre de la figure 4.2, de c en m. Remarquons que le coût correspondant à cette solution est égal à 1 (inférieur au seuil) et, dans A_E, l'état courant (4) est un état final. Nous obtenons ainsi un deuxième candidat de correction.

			a	m
		-2	-1	0
	-2	$(\infty, \top)$	$(\infty, \top$	$(\infty, \top)$
a	-1	$(\infty, \top)$	$(0, \{[]\})$	$(2, \{(add, 0.0, m)(add, t, 0.0.0, g)\})$
c	0	$(\infty, \top)$	$(2, \{(remove, t, 0.0.0, \lambda)(remove, 0.0, \lambda)\})$	$(1, \{(relabel, 0.0, m)\})$

Notons que l'élément $H\_DIST[-1, 0]$ est obtenu en considérant l'arc (1) de la figure 4.3(b). En effet, il correspond au coût permettant de transformer le sous-arbre $\{(0, a)\}$ en sous-arbre $\{(0, a), (0.0, m), (0.0.0, g\}$. L'élément $H\_DIST[0, 0]$ stocke le coût et la distance minimale permettant de transformer

le sous-arbre $\{(0, a),\ (0.0, c),\ (0.0.0, g)\}$ en sous-arbre $\{(0, a),\ (0.0, m),\ (0.0.0, g)\}$. Le choix est fait en considérant les trois cas expliqués dans cette section : selon l'arc (1) de la figure 4.3(b), notre algorithme procède par l'insertion du sous-arbre $\{(0.0, m),\ (0.0.0, g)\}$ après avoir supprimé le sous-arbre $\{(0.0, c), (0.0.0, g)\}$ (comme l'indique l'élément $H\_DIST[0, -1]$). Dans ce cas, le coût est égal 4. Selon l'arc (2) de la figure 4.3(b), notre algorithme procède par le remplacement du sous-arbre $\{(0.0, c),\ (0.0.0, g)\}$ par $\{(0.0, m), (0.0.0, g)\}$. Pour ce faire, il suffit de renommer le nœud c par m. Dans ce cas, le coût est égal à 1. Selon l'arc (3) de la figure 4.3(b), notre algorithme procède par la suppression du sous-arbre $\{(0.0, c), (0.0.0, g)\}$ après avoir inséré le sous-arbre $\{(0.0, m), (0.0.0, g)\}$ (comme l'indique l'élément $H\_DIST[-1, 0]$). Dans ce cas, le coût est égal à 4. □

Les algorithmes qui implémentent cette méthode sont proposés dans [BCHS06b, BCHS06a]. Nous prouvons la correction et la complétude de notre méthode de correction dans [BCHS06a]. Notre méthode est *correcte* puisque tous les candidats T' proposés sont valides et ne dépassent pas le seuil d'erreur th, c'est-à-dire $T' \in \mathcal{L}$ et $dist(T, T') \leq th$. Même si notre méthode n'est pas complète par rapport à l'ensemble de tous les candidats respectant le seuil, elle est *complète* par rapport à l'ensemble des candidats minimaux. Autrement dit, notre méthode est complète puisque chaque candidat minimal qui respecte le seuil th est trouvé par notre algorithme : la routine de correction trouve tous les $T' \in \mathcal{L}$ tels que $dist(T, T') = DIST(T, \mathcal{L})$ si $DIST(T, \mathcal{L}) \leq th$. Remarquer que notre routine de correction calcule non seulement l'ensemble de tous les candidats $t' \in \mathcal{L}$, tels que $dist(t, t')$ est minimale et ne dépasse pas le seuil d'erreur (c'est-à-dire $dist(t, t') = DIST(t, \mathcal{L}) \leq th$), mais également quelques candidats non minimaux qui respectent le seuil.

À partir des corrections locales, différentes solutions globales sont proposées et peuvent être présentées de différentes formes dont la plus intéressante semble être la présentation d'une liste de mise à jour capable de nous donner un arbre mis à jour et correct. Les séquences de mises à jour correctes pour transformer l'arbre original $T0$ en un arbre valide T' sont calculées en utilisant une matrice de distance d'édition entre deux arbres $H\_DIST_{T0,T'}$.

La complexité théorique de notre approche est malheureusement élevée. Dans le pire des cas, la construction des candidats valides par rapport à un langage $\mathcal{L}_{t(\epsilon)}$ est faite en temps $O(n.F_{Max}^{K+th})$ où n est le nombre de nœuds du sous arbre, K est le *fan-out* maximal de l'arbre T et F_{Max} est le *fan-out* maximal de tous les automates à états finis A_E associés aux expressions régulières E du schéma. Par conséquent, la complexité en temps pour corriger toutes les erreurs détectées pendant la validation incrémentale est $O(m.N.F_{Max}^{K+th})$ où m est le nombre de mise à jour et N est le nombre de nœuds de l'arbre entier. Un prototype permettant une étude de performance est en phase d'implémentation. Une performance raisonnable a été obtenue par la méthode de correction *string-to-grammar* proposée dans [CSBH05] et qui sert de base à l'implémentation de notre algorithme de correction des arbres XML.

Travaux liés

Plusieurs approches ont été proposées [Rou03, Sel77, Tai79, ZS89] pour la *correction* d'arbres. En s'inspirant du modèle de calcul de la distance d'édition entre deux chaînes de caractères, la première proposition d'une méthode de calcul de *distance d'édition* entre deux arbres apparaît dans [Sel77]. Des extensions et des optimisations sont proposées dans [Tai79, ZS89]. Notre méthode est une extension naturelle de la comparaison entre deux arbres et propose la correction d'un arbre par rapport à un langage d'arbres.

La correction des arbres XML est aussi traitée dans [Rou03] où les documents XML sont des arbres binaires (voir [Che06] pour un bon résumé de la méthode). La méthode est composée de deux étapes :

1. Validation d'un arbre XML et marquage des nœuds $\star$: Un validateur *from scratch* et *bottom-up* vérifie la structure de l'arbre en marquant avec le symbole $\star$ les positions d'échec. Soit p une position ayant un ou plusieurs descendants marqués par $\star$. Si toutes les substitutions possibles des nœuds $\star$ par des étiquettes appartenant à l'alphabet mènent à une erreur de validation au niveau du nœud p, alors celui-ci est marqué par $\star$. Si le nombre d'erreurs ne dépasse pas un seuil donné, alors la routine de correction est déclenchée.
2. Correction : La correction des nœuds $\star$ est faite de manière *top-down*. À partir de l'arbre marqué avec $\star$, la méthode de correction construit un nouvel arbre en effectuant des modifications (*suppression d'un sous-arbre*, *insertion* ou *renommage d'un nœud*) au voisinage des nœuds marqués par $\star$. Le coût de ces modifications ne doit pas dépasser le seuil d'erreur th.

Une comparaison de performance entre notre méthode et celle de [Rou03] n'a pas encore été effectuée. Même si nous traitons le même problème, notre approche se distingue de celle-ci sur les points suivants :

- Notre méthode fournit différentes solutions dans un seuil d'erreur. En particulier, elle calcule *tous* les candidats dont la distance d'édition par rapport à l'arbre invalide est minimale.
- L' utilisateur peut choisir la solution plus adaptée à son cas.
- Notre méthode fournit aussi une (ou plusieurs) séquence de mises à jour permettant de transformer l'arbre d'origine en candidat valide.
- Notre méthode de correction s'intègre dans un processus de validation incrémentale et peut utiliser des informations de cette procédure dans l'étape de correction.

4.3 Conclusions

Notre routine de correction introduit une approche de correction *tree-to-grammar* : étant donné un langage d'arbre $\mathcal{L}$ (LTL ou $STTL$) et un arbre non valide T, trouver des arbre valides T' dont la distance par rapport à T respecte le seuil th. Ceci est obtenu par le calcul de la distance d'édition entre T et $T' \in \mathcal{L}$. Pour cela, nous considérons le mot w associé aux fils du nœud racine de l'arbre T à corriger. Nous construisons une matrice de distance d'édition afin de déterminer tous les mots w' (corrects) tel que la distance d'édition par rapport w ne dépasse pas le seuil donné. Les caractères $w[i]$ et $w'[j]$ sont les labels de la racine des sous arbres T_i^{sub} (connu) et $T_i'^{sub}$ (à construire). Si $w[i]$ et $w'[j]$ sont différents un appel récursif de la méthode de correction propose des corrections pour que w' soit obtenu.

Notre méthode de correction est associée à la procédure de validation et propose différentes solutions à l'utilisateur. Néanmoins, il est intéressant de remarquer qu'elle peut également être appliqué *from scratch*. Cela peut nous permettre de considérer le calcul de toutes les solutions possibles dans le seuil donné th.

Parmi les perspectives à considérer dans ce domaine, nous pouvons citer (voir chapitre 8) la possibilité d'utilisation d'autres primitives de mises à jour (par exemple l'ajout d'un niveau dans la hiérarchie) et l'étude des corrections commutatives. Remarquer que la commutativité des mises à jour présentée dans le chapitre 3 n'implique pas une commutativité des corrections.

Chapitre 5

Évolution des schémas en XML

Dans ce chapitre nous dissertons sur notre approche où l'évolution d'un schéma XML est déclenchée par des mises à jour sur les documents. Nous considérons des schémas type DTD, c'est-à-dire, correspondant à des grammaires LTG. Cette approche est utilisée dans un environnement d'aide à la maintenance d'une collection de documents XML, par un utilisateur non informaticien, mais connaisseur de son domaine d'application et capable de prendre des décisions sur son évolution [RAJB+00]. Nous offrons à cet utilisateur la possibilité d'engendrer de nouveaux schémas en prenant en compte les caractéristiques du schéma original et des documents mis à jour. Différents choix lui sont proposés, pour qu'il puisse prendre une décision "sémantique " adaptée à ses besoins.

Soit un schéma XML (correspondant à une LTG) représenté par un ensemble d'expressions de la forme $F : a[E]$. Autrement dit, un schéma est une grammaire ayant des règles $F : a[E]$ où F est un non terminal, a est un label et E une expression régulière sur des non terminaux. Les expressions régulières définissent le langage (régulier) des fils d'un nœud (élément) dans un arbre XML. L'évolution du schéma XML correspond ainsi à l'évolution des expressions régulières E et notre problème est formulé de la façon suivante :

> Soient E une expression régulière non ambiguë et w un mot appartenant au langage de E, c'est-à-dire, $w \in L(E)$. Deux types de mises jour sont possibles sur le mot w : l'insertion ou la suppression d'un symbole, sur une position de mise à jour i de w. Soit w' le mot obtenu suite à *une* mise à jour sur w. Si $w' \notin L(E)$, alors nous supposons l'activation d'une procédure qui engendre des expressions E' proches de E telles que $w' \in L(E')$, $L(E) \subseteq L(E')$.

Les langages $L(E')$ ne doivent pas seulement contenir w' mais tous les mots sémantiquement proches de w'. Pour cela, dans nos travaux, une notion très simple de distance entre deux expressions régulières a été introduite. Les solutions proposées respectent une distance minimale par rapport à l'expression régulière d'origine. L'objectif du travail est de fournir différents choix d'expressions régulières, en essayant de prévoir les besoins de l'application tout en restant proche de l'expression régulière originale. Ainsi, chaque expression régulière candidate E' correspond à un langage $L(E')$ plus général que $L(E) \cup \{w'\}$. Nous ne sommes pas intéressés par le candidat $E + w'$ qui ajoute un seul mot à $L(E)$, ni par des candidats trop généraux permettant n'importe quel type de mise à jour.

Deux versions de cette approche ont été développées. Une première version est proposée dans la thèse de Denio Duarte [Dua05], que j'ai co-encadré. Dans cette version, les extensions sur les expressions régulières sont obtenues par un algorithme (nommé `GREC`) qui change *l'automate d'états fini M_E associé à chaque expression E*. L'algorithme `GREC` est une extension de la procédure de réduction proposée par [CZ00] pour transformer un automate de Glushkov en expression régulière. Une version de `GREC` sans l'utilisation des automates d'états fini M_E a été proposée dans [dHM07]. Cette deuxième version, que nous appelons `dGREC`[1], obtient les mêmes solutions que `GREC` en travaillant seulement sur les expressions régulières, sans passer par la transformation des expressions régulières en automates. Pour cela des fonctions introduites dans [ZPC97] sont utilisées.

[1] La méthode `dGREC` fait partie des travaux de *Master of Science* de Robson da Luz (encadré par Martin Musicante, au Brésil) [dL07].

Ce chapitre est consacré à une discussion sur ces différentes versions de notre approche pour l'évolution de schéma. D'abord, dans la section 5.1 nous présentons un rappel rapide de certains concepts, nécessaires pour la compréhension du chapitre. Ensuite, dans les sections 5.2 et 5.3, nous présentons les principes et les algorithmes généraux pour `GREC` et `dGREC`. Dans l'annexe B, nous expliquons plus en détail les étapes de transformation des expressions régulières sur les deux approches `GREC` et `dGREC`. Cela nous permet de faire un parallèle entre les deux versions qui ont été publiées séparément pour ensuite montrer que les versions sont équivalentes. Une discussion sur les similarités et différences des deux approches est présentée dans la section 5.4.

Dans [BDH+04a, BDH+04b, Dua05] la validation d'un document XML est faite via un automate d'arbres. L'évolution du schéma représenté par cet automate correspond au choix d'une expression régulière E' proposée par `GREC` ou `dGREC`. Dans [dHM07] la validation d'un document XML est faite par l'exécution d'un programme datalog et non par un automate d'arbres. Dans la section 5.5 nous présentons une fonction de traduction du schéma XML en datalog et ensuite nous introduisons `dGREC`^{Dat}, dont la définition est similaire à celle de `dGREC`, pour effectuer l'évolution incrémentale du validateur datalog. La section 5.8 conclut ce chapitre après une discussion sur les extensions possibles de `GREC` et `dGREC` (section 5.6) et sur l'état de l'art (section 5.7).

5.1 Préliminaires

Tout d'abord, nous rappelons brièvement des notions et des résultats très connus concernant les automates d'états finis et les expressions régulières. Une référence (classique) pour une introduction dans le domaine est [HMU01].

Les expressions rationnelles ou régulières sont une famille de notations compactes et puissantes pour décrire des langages réguliers. Étant donné un alphabet $\Sigma = \{a_1, \ldots, a_n\}$, nous rappelons que l'ensemble des expressions régulières sur Σ est inductivement défini par : $E ::= \emptyset \mid \epsilon \mid a_i \mid E+E \mid E.E \mid E^+ \mid E^* \mid E? \mid (E)$. Chaque expression régulière définit un langage inductivement spécifié par :

$L(\emptyset)$	$= \emptyset$
$L(\epsilon)$	$= \{\epsilon\}$
$L(a_i)$	$= \{a_i\}$
$L(E.F)$	$= L(E).L(F)$ où E et F sont des expressions régulières et . indique la concaténation des langages. Nous appelons . l'opérateur de concaténation.
$L(E+F)$	$= L(E) \cup L(F)$ où E et F sont des expressions régulières.
$L(E^*)$	$= L(E)^*$ où E est une expression régulière et * est la fermeture de Kleene.

Un automate d'états fini (*finite state automaton* (FSA)) est une machine abstraite constituée d'états et de transitions. Son comportement est dirigé par un mot fourni en entrée : l'automate passe d'état en état, suivant les transitions, à la lecture de chaque lettre de l'entrée. Un automate d'états fini forme naturellement un graphe orienté étiqueté, dont les états sont les sommets et les transitions les arêtes étiquetées (par exemple, la figure 5.1(a)). Un automate d'états fini est défini par le n-uplet $M = (S, \Sigma, \delta, s_0, F)$ où (i) S est un ensemble fini d'états ; (ii) Σ est l'alphabet de M, c'est-à-dire, un ensemble fini des symboles d'entrée ; (iii) $s_0 \in S$ est l'état initial de M ; (iv) $F \subseteq S$ est l'ensemble d'états finaux et (v) δ est une fonction de transition. La fonction δ prend comme arguments d'entrée un état dans S et un symbole dans Σ. Dans un automate d'états fini *déterministe*, δ rend comme résultat un état de S, alors que dans un automate d'états fini *non déterministe*, δ rend comme résultat un sous ensemble de S. Nous notons $\delta(1, b) = 2$ pour indiquer la transition de l'état 1 à l'état 2 en lisant b (figure 5.1(a)).

Si $M = (S, \Sigma, \delta, s_0, F)$ est un automate d'états fini alors $L(M)$ est le langage accepté par M, c'est-à-dire, $L(M)$ est l'ensemble de mots w dans Σ^* tel que, à partir de l'état s_0, en lisant le mot w, nous arrivons à au moins un état final.

Tout langage qui peut être décrit par un automate non déterministe peut aussi être décrit par un automate déterministe et vice versa. En autres termes, il existe une équivalence entre les automates d'états finis déterministes et non déterministes. Tout langage défini par un automate d'états fini est aussi défini par une expression régulière et vice versa. Ainsi, étant donnée une expression régulière E, nous notons M_E l'automate d'états fini tel que $L(E) \equiv L(M_E)$ (la figure 5.1(a) montre un automate correspondant à l'expression régulière $(a.(b+c)^*)^*.d)$. Dans nos travaux nous considérons les expressions régulières non

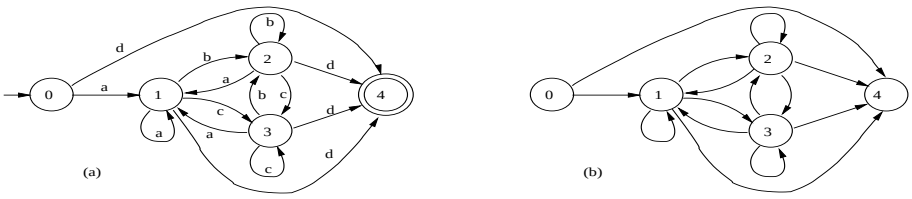


FIG. 5.1: (a) Un automate d'états fini pour $(a_1.(b_2 + c_3)^*)^*.d_4$. (b) Graphe de Glushkov correspondant.

ambiguës. Une expression régulière E est non ambiguë si pour chaque mot w il existe au maximum un chemin dans E qui peut être associé à w [BW98a].

Étant donnée une expression E sur un alphabet Σ, nous définissons l'expression régulière souscrite $\overline{E}$ comme étant l'expression régulière où chaque symbole de E a un indice (aussi appelé position) unique. Par exemple, soit $E = (a.(b + c)^*)^*.d$, l'expression régulière avec des indices indiquant les positions est $\overline{E} = (a_1.(b_2 + c_3)^*)^*.d_4$. Remarquer que, comme un symbole est identifié par la position, il peut être omis, c'est-à-dire, nous pouvons réécrire l'expression régulière avec les positions seulement. Dans ce chapitre, par abus de notation, nous allons écrire (souvent) E à la place de $\overline{E}$. Nous faisons référence au symbole associé à une positon p par $symb_E(p)$. Par exemple, nous pouvons réécrire l'expression régulière E comme $(1.(2 + 3)^*)^*.4$ en prenant en compte que $symb(1) = a$, $symb_E(2) = b$, $symb_E(3) = c$ et $symb_E(4) = d$. L'ensemble des positions d'une expression régulière E est noté $Pos(E)$. Remarquer que dans l'exemple nous avons choisi de nommer les positions dans l'ordre (de gauche à droite), en utilisant les nombres naturels. Néanmoins cela n'est pas une obligation ; n'importe quel nom peut être utilisé dès qu'il est un identifiant unique de la position dans l'expression régulière. Étant donné un mot w, nous définissons $|w|$ comme étant la longueur du mot w, par exemple si $w = abc$ alors $|w| = 3$.

Dans la suite, la section 5.1.1 rappelle les caractéristiques des automates de Glushkov qui sont utilisés par `GREC`. La section 5.1.2 présente des fonctions très utiles sur les expressions régulières. Ces fonctions ont été introduites dans [ZPC97] et sont utilisées dans la version `dGREC`.

5.1.1 Automate de Glushkov

Un automate de Glushkov ([CZ00]) est un automate d'états fini homogène où chaque état non initial correspond à une position dans l'expression régulière. Un automate $M_E = (S, \Sigma, \delta, s_0, F)$ est homogène si $\forall p, q, r \in S$, nous avons : $\exists a, b \in \Sigma$, $((\delta(p, a) = q) \wedge (\delta(r, b) = q)) \Rightarrow a = b$. En d'autres termes, un état est toujours atteint par le même symbole.

Le *graphe de Glushkov* correspondant à un automate de Glushkov donné est un graphe $G = (X, U)$ où X est l'ensemble des nœuds (isomorphe à l'ensemble d'état de M_E) et U est l'ensemble des arcs (correspondant aux transitions de δ). Pour les automates homogènes, il est possible d'ignorer les labels des arcs et travailler avec un graphe orienté (voir figure 5.1).

Selon le théorème dans [CZ00], un graphe $G = (X, U)$ est un graphe de Glushkov ssi les trois conditions suivantes sont respectées :

1 - G est un hamac
Un graphe est appelé *hamac* s'il a un noeud racine r n'ayant aucun arc entrant et un noeud anti-racine s n'ayant aucun arc sortant, tels que $r \neq s$. Un graphe a un noeud *racine* (respectivement un noeud *anti-racine*) r s'il existe un chemin de r à n'importe quel noeud (respectivement de n'importe quel noeud à s) dans le graphe.

2 - Chaque orbite maximale dans G est fortement stable et fortement transversale
Étant donné un graphe de Glushkov $G = (X, U)$, une *orbite* est un ensemble de nœuds tel que pour tout x et x' de $\mathcal{O}$, il existe un chemin non trivial[2] de x à x'. Une orbite est *maximale* si pour tout noeud x qui appartient à $\mathcal{O}$ et pour tout noeud x' qui n'appartient pas à $\mathcal{O}$, il n'existe pas à la fois un chemin de x à x' et de x' à x. Les portes d'entrée (In) et de sortie (Out) d'une orbite $\mathcal{O}$ sont définies de la façon

[2]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 trivial est un chemin sans aucun arc.

suivante $In(\mathcal{O}) = \{x \in \mathcal{O} \mid \exists x' \in (X \setminus \mathcal{O}), (x', x) \in U\}$ et $Out(\mathcal{O}) = \{x \in \mathcal{O} \mid \exists x' \in (X \setminus \mathcal{O}), (x, x') \in U\}$. Une orbite est *stable* si $\forall x \in Out(\mathcal{O})$ et $\forall y \in In(\mathcal{O})$, l'arc (x, y) existe. Une orbite $\mathcal{O}$ est *transversale* si $\forall x, y \in Out(\mathcal{O})$, $\forall z \in (X \setminus \mathcal{O}), (x, z) \in U \Rightarrow (y, z) \in U$ et $\forall x, y \in In(\mathcal{O})$, $\forall z \in (X \setminus \mathcal{O}), (z, x) \in U \Rightarrow (z, y) \in U$. Une orbite $\mathcal{O}$ est *fortement stable* (respect. *fortement transversale*) si elle est stable (respect. transversale) et si, après la suppression des arcs $Out(\mathcal{O}) \times In(\mathcal{O})$, toutes les orbites résultantes sont fortement stables (respect. fortement transversales).

3 - G_{wo} est réductible.
Étant donné un graphe de Glushkov G, un *graphe sans orbites* G_{wo} est défini, récursivement, en supprimant pour chaque orbite maximale $\mathcal{O}$, tous les arcs (x, y) 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. Étant donné un G_{wo}, nous disons qu'il est réductible s'il est possible de le transformer en un seul nœud via l'application des règles des réductions introduites dans [CZ00] et présentées dans la section 5.2.1. À chaque étape de cette transformation certains nœuds de G_{wo} sont "fusionnés" et la réduction est complète à l'obtention d'un seul nœud.

Exemple 5.1.1 La figure 5.1(b) présente le graphe de Glushkov G associé à l'automate de Glushkov de la figure 5.1(a). G possède une orbite maximale $\mathcal{O}_1 = \{1, 2, 3\}$ (avec $In(\mathcal{O}_1) = \{1\}$ et $Out(\mathcal{O}_1) = \{1, 2, 3\}$). L'orbite $\mathcal{O}_1$ est transversale et stable. Nous pouvons construire un graphe sans orbites à partir de G de la façon suivante : (i) Supprimer les arcs $Out(\mathcal{O}_1) \times In(\mathcal{O}_1)$ de G. (ii) Le graphe résultant G' possède aussi une orbite maximale $\mathcal{O}_2 = \{2, 3\}$ (avec $In(\mathcal{O}_2) = Out(\mathcal{O}_2) = \{2, 3\}$). Supprimer les arcs $Out(\mathcal{O}_2) \times In(\mathcal{O}_2)$ pour obtenir le nouveau graphe G'' sans orbites. Les deux orbites maximales $\mathcal{O}_1$ et $\mathcal{O}_2$ sont fortement stables et fortement transversales. Le graphe G_{wo} est celui de la figure 5.3(a). Pendant la construction de G_{wo} les informations concernant les orbites éliminées sont stockées, c'est-à-dire, nous gardons trace des orbites de G. □

5.1.2 Fonctions sur les expressions régulières

Nous présentons ici des fonctions très utiles sur les expressions régulières. Ces fonctions ont été introduites dans [ZPC97].

Définition 5.1.1 - La fonction $Null_E$: $Null_E$ est égal à $\{\epsilon\}$ si ϵ est dans $L(E)$, sinon $Null_E$ est $\emptyset$. L'opération $Null_E$ est inductivement définie de la façon suivante.

$$Null_\emptyset = \emptyset \qquad Null_\epsilon = \{\epsilon\} \qquad Null_a = \emptyset$$
$$Null_{F+G} = Null_F \cup Null_G \qquad Null_{F.G} = Null_F \cap Null_G \qquad Null_{F^+} = Null_F$$
$$Null_{F^*} = \{\epsilon\} \qquad Null_{F?} = \{\epsilon\} \qquad Null_{(E)} = Null_E \quad \square$$

La fonction $Null_E$ peut être calculée en temps linéaire sur la longueur de E.

Dans la suite, nous denotons $\overline{\Sigma}$ l'alphabet souscrit de E.

Définition 5.1.2 - La fonction $First(E)$: $First(\overline{E})$ est défini comme un ensemble de positions dans $\overline{E}$, correspondant à la première lettre d'un mot de $L(E)$.

$$First(\overline{E}) = \{x \in Pos(E) \mid \exists u \in \overline{\Sigma}^* : \alpha_x u \in L(\overline{E})\}$$

RAPPEL : Cette fonction est définie sur une expression régulière souscrite. Par abus de notation nous écrivons $First(E)$ pour $First(\overline{E})$. La même convention sera utilisée pour les autres fonctions.

La fonction $First(E)$ peut être calculée inductivement de la façon suivante

$$First(\emptyset) = \emptyset \qquad First(\epsilon) = \emptyset$$
$$First(\alpha_x) = \{x\} \qquad First(F + G) = First(F) \cup First(G)$$
$$First(F.G) = First(F) \cup Null_F.First(G) \qquad First(F^+) = First(F)$$
$$First(F^*) = First(\ (F)\ ) = First(F) \qquad First(F?) = First(F) \qquad \square$$

Dans les équations ci-dessus, le calcul de l'union est fait en temps linéaire. Cela est possible parce que les ensembles sont disjoints (ils correspondent à des ensembles de positions dans différentes parties d'une expression régulière). Ainsi $First(E)$ peut être calculée en temps linéaire.

Définition 5.1.3 - La fonction $Last(E)$ **:** Étant donnée une expression régulière E, nous définissons $Last(E)$ comme un ensemble des positions correspondant aux symboles finales des mots du langage $L(E)$.

$$Last(E) = \{x \in Pos(E) \mid \exists u \in \overline{\Sigma}^* : u\alpha_x \in L(\overline{E})\}$$

La fonction $Last(E)$ est inductivement calculée de la façon suivante :

$$\begin{aligned} Last(\emptyset) &= \emptyset & Last(\epsilon) &= \emptyset \\ Last(\alpha_x) &= \{x\} & Last(F+G) &= Last(F) \cup Last(G) \\ Last(F.G) &= Last(G) \cup Null_G.Last(F) & Last(F^+) &= Last(F) \\ Last(F^*) &= Last(\ (F)\ ) = Last(F) & Last(F?) &= Last(F) \end{aligned}$$ □

Comme dans la définition précédente, l'union peut être calculée en temps linéaire. Ainsi, $Last(E)$ est calculée en temps linéaire par rapport à la longueur de E.

Pour chaque ensemble X, nous dénotons $\mathcal{I}_X$ la fonction de X à $\{\{\epsilon\}, \emptyset\}$ telle que :

$$\mathcal{I}_X(x) = \emptyset \text{ if } x \notin X \qquad \text{and} \qquad \mathcal{I}_X(x) = \{\epsilon\} \text{ if } x \in X.$$

Cette fonction est utilisée dans les définitions suivantes, pour sélectionner des positions d'une expression régulière.

Définition 5.1.4 - La fonction $Follow(E, x)$ **:** $Follow(E, x)$ est l'ensemble de positions immédiatement après la position x dans l'expression E. Si x n'est pas dans l'ensemble de positions de E alors $Follow(E, x) = \emptyset$.

$$Follow(E, x) = \{y \in Pos(E) \mid \exists v, w \in \overline{\Sigma}^* : v\alpha_x\alpha_y w \in L(\overline{E})\}$$

La fonction $Follow(E, x)$ peut être inductivement calculée de la façon suivante :

$$\begin{aligned} Follow(\emptyset, x) &= Follow(\epsilon, x) = Follow(a, x) = \emptyset \\ Follow(F+G, x) &= \mathcal{I}_{Pos(F)}(x).Follow(F, x) \cup \mathcal{I}_{Pos(G)}(x).Follow(G, x) \\ Follow(F.G, x) &= \mathcal{I}_{Pos(F)}(x).Follow(F, x) \cup \mathcal{I}_{Pos(G)}(x).Follow(G, x) \\ &\quad \cup \mathcal{I}_{Last(F)}(x).First(G) \\ Follow(F^+, x) &= Follow(F, x) \cup \mathcal{I}_{Last(F)}(x).First(F) \\ Follow(F^*, x) &= Follow(F, x) \cup \mathcal{I}_{Last(F)}(x).First(F) \\ Follow(F?, x) &= Follow(\ (F)\ , x) = Follow(F, x) \end{aligned}$$ □

Définition 5.1.5 - La fonction $Previous(E, x)$ **:** $Previous(E, x)$ est un ensemble de positions immédiatement avant la position x dans l'expression E. Si x n'est pas dans l'ensemble de positions de E alors $Previous(E, x) = \emptyset$.

$$Previous(E, x) = \{y \in Pos(E) \mid \exists v, w \in \overline{\Sigma}^* : v\alpha_y\alpha_x w \in L(\overline{E})\}$$

Le calcul de $Previous(E, x)$ est similaire au calcul de $Follow(E, x)$. La fonction $Previous(E, x)$ peut être inductivement calculée de la façon suivante :

$$\begin{aligned} Previous(\emptyset, x) &= Previous(\epsilon, x) = Previous(a, x) = \emptyset \\ Previous(F+G, x) &= \mathcal{I}_{Pos(F)}(x).Previous(F, x) \cup \mathcal{I}_{Pos(G)}(x).Previous(G, x) \\ Previous(F.G, x) &= \mathcal{I}_{Pos(F)}(x).Previous(F, x) \cup \mathcal{I}_{Pos(G)}(x).Previous(G, x) \\ &\quad \cup \mathcal{I}_{First(G)}(x).Last(F) \\ Previous(F^+, x) &= Previous(F, x) \cup \mathcal{I}_{First(F)}(x).Last(F) \\ Previous(F^*, x) &= Previous(F, x) \cup \mathcal{I}_{First(F)}(x).Last(F) \\ Previous(F?, x) &= Previous(\ (F)\ , x) = Previous(F, x) \end{aligned}$$ □

Toutes les fonctions spécifiées par les définitions 5.1.1 à 5.1.5 peuvent être implémentées en temps constant ou en temps linéaire, en utilisant des opérations ensemblistes.

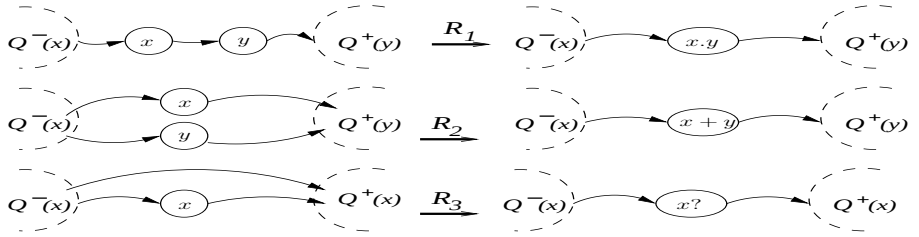


FIG. 5.2: Règles de réduction.

5.2 Transformation des expressions régulières via des automates de Glushkov

Soit une spécification de schéma contenant des expressions de la forme $F : a[E]$. Dans cette section, nous considérons une méthode de transformation des expressions régulières composée de trois grandes étapes, à savoir :
(*i*) La construction d'un automate à états fini M_E reconnaissant $L(E)$.
(*ii*) La modification de M_E pour obtenir un nouvel automate M'_E qui accepte le nouveau mot w'.
(*iii*) L'obtention de E' à partir de M'_E.

Cette méthode de transformation des expressions régulières a été proposée dans [BDH$^+$04b, BDH$^+$04a, Dua05] et est fondée sur la procédure de traduction d'un automate de Glushkov en une expression régulière équivalente présentée dans [CZ00]. Dans la section 5.2.1 nous rappelons cette procédure alors que la section 5.2.2 résume l'algorithme `GREC` proposé pour modifier M_E en produisant des nouvelles expressions régulières E'.

5.2.1 Procédure de transformation d'un automate de Glushkov en expression régulière

Soit G un graphe de Glushkov. La procédure de réduction proposée dans [CZ00] a comme but obtenir l'expression régulière E correspondant à G. Pour cela l'algorithme de réduction utilise le graphe sans orbites G_{wo}. La procédure débute en considérant que chaque nœud de G_{wo} est associé à une expression régulière correspondant au nom d'un état de G (ou de l'automate de Glushkov). Ensuite la procédure consiste à : (i) détecter des nœuds (x ou y) qui respectent certaines conditions ; (ii) remplacer les nœuds détectés par un nouveau nœud (représentant la fusion des nœuds x et y) et (iii) associer une nouvelle expression régulière (par exemple $x.y$ ou $x + y$) à ce nouveau nœud. La réduction termine quand G_{wo} est un graphe avec un unique nœud auquel est associé l'expression régulière E.

Les conditions à vérifier sur les nœuds x et y concernent leurs prédécesseurs et successeurs. La définition ci-dessous introduit cette notion.

Définition 5.2.1 - Ensemble des nœuds prédécesseurs et des nœuds successeurs dans un graphe sans orbite : Soit $G_{wo} = (X, U)$ un graphe sans orbites. Soit x un noeud dans G_{wo}. Définir $Q^-(x) = \{y \in X \mid (y, x) \in U\}$ comme l'ensemble de prédécesseurs immédiats de x et $Q^+(x) = \{y \in X \mid (x, y) \in U\}$ l'ensemble de successeurs immédiats de x. □

Étant donné G_{wo}, nous disons qu'il est *réductible* ([CZ00]) s'il est possible de le réduire à un nœud (état) en appliquant successivement l'une des trois règles $\mathbf{R}_1$, $\mathbf{R}_2$ et $\mathbf{R}_3$ illustrées par la figure 5.2. Les règles de réduction sont définies ci-dessous (nous notons $r(x)$ l'expression régulière associée au nœud x, et e l'expression régulière résultante dans chaque cas) :

Règle $\mathbf{R}_1$: S'il existe deux nœuds x et y tels que $Q^-(y) = \{x\}$ et $Q^+(x) = \{y\}$, c'est-à-dire, le noeud x est le seul prédécesseur du noeud y et le noeud y est le seul successeur du noeud x, alors concaténer $r(x)$ et $r(y)$ en e, attribuer e à x et supprimer y.

Règle $\mathbf{R}_2$: S'il existe deux noeuds x et y tels que $Q^-(x) = Q^-(y)$ et $Q^+(x) = Q^+(y)$, c'est-à-dire, les noeuds x et y ont les mêmes prédécesseurs et les mêmes successeurs, alors construire e qui correspond à l'union de $r(x)$ et $r(y)$, attribuer e à x et supprimer y.

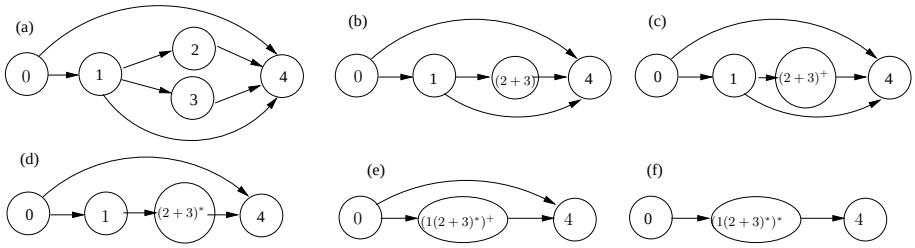


FIG. 5.3: Un exemple de réduction.

Règle $\mathbf{R}_3$: S'il existe un noeud x tel que $y \in Q^-(x) \Rightarrow Q^+(x) \subset Q^+(y)$, c'est-à-dire, le noeud x est optionnel, alors supprimer les arcs de $Q^-(x)$ à $Q^+(x)$ et construire une expression régulière comme suit : si $r(x)$ est de la forme E^+ (respectivement E) alors la nouvelle expression e est E^* (respectivement $E?$).

Pendant la construction de G_{wo}, les orbites sont ordonnées par rapport à l'inclusion ensembliste (algorithme dans [Dua05]) pour constituer une hiérarchie d'orbites $\mathcal{H}$. La procédure de réduction commence par l'orbite la plus interne dans la hiérarchie $\mathcal{H}$ de G_{wo}. La réduction commence au niveau le plus bas de la hiérarchie (des orbites les plus internes aux orbites les plus englobantes).

Exemple 5.2.1 Étant donné $E = a_1.(b_2^*.c_3)^*.(d_4.e_5)^+.f_6$, la hiérarchie des orbites $\mathcal{H}$ est représentée par un n-uplet, c'est-à-dire, $\mathcal{H} = \langle\{2\}, \{2,3\}, \{4,5\}, \{1,2,3,4,5,6\}\rangle$. En effet, $\mathcal{H}$ peut être vu comme un arbre où la racine correspond à tous les nœuds du graphe. Ses fils sont les orbites incluses dans la racine. Dans notre exemple, la racine correspond à $\{1,2,3,4,5,6\}$ et possède deux fils, à savoir, les orbites $\{2,3\}$ (ayant l'orbite $\{2\}$ comme fils) et $\{4,5\}$. □

L'information concernant les orbites dans le graphe original est utilisée pour ajouter l'opérateur de fermeture transitive ("$^+$") à l'expression régulière en construction. Pendant la procédure de réduction, quand un nœud unique représente une seule orbite, son contenu est décoré avec un '$^+$". Dans nos travaux, cette opération est appelée "application de la règle $\mathbf{R}_4$.

Exemple 5.2.2 La figure 5.1 montre un automate et le graphe de Glushkov correspondant. Soit G_{wo} le graphe sans orbites correspondant (figure 5.3(a)). La hiérarchie des orbites maximales est $\mathcal{H}$=$\langle\{2,3\}$, $\{1,2,3\}$, $\{0,1,2,3,4\}\rangle$. Soient $\mathcal{O}_3 = \{2,3\}$ et $\mathcal{O}_1 = \{1,2,3\}$. Les autres orbites du graphe ne sont pas dans la hiérarchie car, pendant le processus de construction de G_{wo}, elles ne sont pas maximales.

La figure 5.3 illustre des étapes du processus de réduction. La figure 5.3(b) est le résultat de l'application de la règle $\mathbf{R}_2$. Le noeud (2+3) représente l'orbite $\mathcal{O}_3$. Dans ce cas, nous construisons la fermeture positive $(2+3)^+$ (figure 5.3(c), application de $\mathbf{R}_4$). Après l'application de la règle $\mathbf{R}_3$, l'expression $(2+3)^+$ devient optionnelle (figure 5.3(d)). En appliquant la règle $\mathbf{R}_1$ et en prenant en considération l'orbite $\mathcal{O}_1$, le graphe de la figure 5.3(e) est obtenu. La figure 5.3(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 correspond à l'état initial et chaque position correspond au symbole dans l'expression régulière d'origine). □

REMARQUES :

• Comme dans [CZ00] le symbole # est ajouté à la fin des expressions régulières de façon à assurer les propriétés du graphe associé à l'automate pour caractériser un automate de Glushkov.

• Nous utilisons la notation $E!$ comme une abréviation, pour représenter $E?$ ou E^*.

5.2.2 Évolution des expressions régulières : l'algorithme `GREC`

L'algorithme `GREC` (**G**enerate **R**egular **E**xpression **C**hoices) construit les expressions régulières candidates, à partir de la méthode de réduction proposée par [CZ00]. En effet, avant l'application de chaque règle de réduction, `GREC` teste si une modification sur l'automate peut être faite. Les modifications sont guidées par des conditions à vérifier sur deux états de M_E définis à partir de l'exécution de M_E sur w :

> Nous rappelons que le mot $w' \notin L(E)$ est obtenu à partir d'une *unique* mise à jour sur le mot $w \in L(E)$. Soit p $(0 \leq p \leq |w|)$ la position de mise à jour en w (une suppression ou une insertion).

```
01. function GREC(G_wo, H, s_nl, s_nr, s_new){
02.   if G_wo has only one node
03.      then return
04.      else{
05.        R_i := ChooseRule(G_wo , H) ;
06.        // the following lines extend the original algorithm GraphToRegExp
07.        for each (G_new, H_new) := LookForGraphAlternative(G_wo ,H,R_i ,s_nl ,s_nr ,s_new) do
08.            return GraphToRegExp(G_new , H_new) ;
09.        // end of extension
10.        (G'_wo ,H_1) := ApplyRule(R_i , G_wo , H) ;
11         return GREC(G'_wo , H_1 , s_nl , s_nr , s_new) ;
12.      }     }
```

FIG. 5.4: L'algorithme pour calculer les expressions régulières candidates.

- L'état s_{nl} (*the nearest left state*) est l'état de M_E atteint en lisant le $(p-1)$ème symbole de w.
- L'état s_{nr} (*the nearest right state*) est l'état de M_E atteint en lisant le $(p+1)$ème symbole de w (si la mise à jour est une suppression) ou le le p-ème symbole de w (si la mise à jour est une insertion).
- L'état s_{new} est un nouvel état.

Remarquez que l'état s_{nl} existe toujours dans M_E car le mot w appartient au langage accepté par M_E. Si M_E est déterministe alors les états s_{nl} et s_{nr} sont uniques.

Nous présentons `GREC` en considérant les insertions seulement. Le cas des suppressions est plus simples et sera considéré dans l'annexe B.0.5. Sans perte de généralité, nous pouvons considérer qu'une insertion correspond toujours à l'insertion d'un nouveau symbole dans l'expression régulière E (même si le symbole existe déjà dans E, car il s'agit de l'insertion d'un position dans E). Ainsi, pour accepter un nouveau mot, il faut ajouter à M_E un nouvel état s_{new} et une transition de s_{nl} à s_{new} et faire les modifications nécessaires pour maintenir les propriétés d'un graphe Glushkov. Ces modifications dépendent des caractéristiques des positions de s_{nl} et s_{nr} dans le graphe.

La figure 5.4 présente un algorithme pour le calcul de la fonction `GREC`. L'algorithme prend cinq paramètres : un graphe sans orbites G_{wo} (associé à l'automate de Glushkov M_E), la hiérarchie des orbites $\mathcal{H}$ (calculée pendant la construction de G_{wo}), les noeuds s_{nl}, s_{nr} et s_{new}. Chaque étape de la procédure de [CZ00] remplace une partie du graphe par un nœud contenant une expression plus complexe. Dans `GREC` nous devons non seulement réduire le graphe en une expression régulière mais aussi faire des changements à chaque fois qu'un nœud correspondant à s_{nl} ou s_{nr} vérifie certaines conditions.

Dans la figure 5.4, `GREC` choisi une règle de réduction en utilisant les informations sur les orbites (fonction `ChooseRule`). Ensuite, deux actions différentes sont exécutées :

1. Avant l'application de la règle R_i choisie, `GREC` teste si les noeuds s_{nr} ou s_{nl} sont affectés par R_i. Quand c'est le cas, `GREC` change le graphe de façon à prendre en compte l'insertion du nœud s_{new}. Ces changements sont guidés par les règles $\mathbf{R}_1$, $\mathbf{R}_2$ et $\mathbf{R}_3$ ainsi que par les informations sur les orbites du graphe original. Chaque modification est effectuée par la fonction `LookForGraphAlternative`, ayant un double rôle :
   (a) vérifier si les nœuds s_{nl} et s_{nr} satisfont les conditions établies par les règles $\mathbf{R}_1$, $\mathbf{R}_2$ ou $\mathbf{R}_3$ et
   (b) engendrer des nouvelles données (un nouveau graphe G_{new} et sa hiérarchie d'orbites $\mathcal{H}_{new}$) sur lesquelles la procédure de réduction [CZ00] originale (`GraphToRegExp`) est appliquée pour fournir des expressions régulières candidates (une expression régulière pour chaque G_{new}).
2. La fonction `ApplyRule` calcule un nouveau graphe par application de R_i sur le G_{wo} original. `GREC` est appliqué récursivement à ce nouveau graphe. Le processus s'arrête lorsque G_{wo} est un seul noeud.

Les conditions associées à chaque règle de réduction sont présentées dans l'annexe B. Nous illustrons le fonctionnement de la méthode via un exemple qui montre pas à pas comment trouver des expressions régulières candidates. Les résultats de cet exemple sont présentés dans la figure 5.5 qui illustre les deux actions de `GREC` décrites ci-dessus. La partie gauche de la figure montre la procédure de réduction qui

commence avec le graphe G_{wo}. La partie droite montre des graphes G_{new} obtenus par l'insertion de s_{new}. À chaque étape de la procédure de réduction où les conditions testées par `GREC` sont satisfaites, plusieurs graphes G_{new} peuvent être construits.

Exemple 5.2.3 Nous considérons l'expression $E = a.(b + c)?.d^+.\#$ qui nous noterons simplement $E = 1.(2 + 3)?.4^+.5$. Soient les mots $w = abd \in L(E)$ et $w' = abnd \notin L(E)$. Nous avons donc $s_{nl} = 2$ et $s_{nr} = 4$. La hiérarchie des orbites est $\mathcal{H} = \{\{1, 2, 3, 4, 5\}, \{4\}\}$.

L'algorithme commence par l'orbite maximale la plus interne, dans notre cas $\{4\}$. Comme $s_{nr} = 4$ est un état correspondant à une orbite, les conditions concernant la situation des nœuds vis à vis des orbites (annexe B, figure B.4) sont à tester (nous appelons cela les tests de la règle $\mathbf{R}_4$). L'état s_{nr} est une orbite et l'erreur dû à l'insertion est constatée avant l'entrée dans cette orbite (condition 2 dans la figure B.4) : s_{nl} est un état prédécesseur de celui correspondant à l'orbite (remarquer que l'ensemble de prédécesseurs de 4 est $\{1, 2, 3, 4\}$). Les solutions proposées ajoutent le nouvel état s_{new} à l'orbite en assurant qu'il soit entre s_{nl} et s_{nr}, c'est-à-dire, dans ce cas, s_{new} doit être une porte d'entrée de l'orbite. Nous obtenons ainsi les deux premières solutions illustrées à droite de la figure 5.5, à savoir : $E' = 1.(2 + 3)?.(6!.4)^+.5$ où $s_{new} = 6$ est inséré dans l'orbite via une concaténation qui précède 4 et $E' = 1.(2+3)?.(6! + 4)^+.5$ où s_{new} est inséré dans l'orbite via une disjonction.

Des conditions liées à règle $\mathbf{R}_2$ sont aussi vérifiées. Le nœud s_{nl} est un des nœuds de la disjonction $(2+3)?$ alors que s_{nr} est un des successeurs de 2. Dans ce cas, la solution proposée (annexe B, figure B.2) ajoute le nouvel état dans la suite de 2, c'est-à-dire, l'expression $E' = 1.(2.6! + 3)?.4^+.5$ est proposée (figure 5.5).

Comme aucune autre condition est vérifiée, l'algorithme continue en appliquant la règle $\mathbf{R}_1$ pour réduire le graphe original. Cela correspond aux deuxième graphe à gauche de la figure 5.5. Aucune condition est vérifiée sur ce deuxième graphe et, ainsi, la réduction continue avec l'application de la règle $\mathbf{R}_3$ pour obtenir le troisième graphe à gauche de la figure 5.5. À ce moment là des conditions associées à la règles $\mathbf{R}_1$ sont vérifiées (annexe B, figure B.1). En effet, $s_{nl} = 2$ est un état qui apparaît dans l'expression $(2+3)?$ du troisième graphe de la réduction alors que $s_{nr} = 4$ est successeur de 2. La solution proposée consiste à ajouter $s_{new} = 6$ entre 2 et 4 ans ajouter s_{new} à l'orbite. En d'autres termes, $E' = 1.(2 + 3)?.6!.4^+.5$ (annexe B, figure B.1).

La procédure de réduction continue, mais aucune autre condition permettant des changements du graphe est vérifiée. L'algorithme s'arrête quand l'automate original est réduit en un seul nœud associé à l'expression E original.

Dans la figure 5.5 nous indiquons les règles dont les conditions associées par `GREC` ont été vérifiées. Chaque G_{new} est transformé par la fonction `GraphToRegExp` en expression régulière candidate. La fonction `GraphToRegExp` implémente la procédure de réduction originale de [CZ00]. Dans cette exemple, l'algorithme nous propose quatre nouvelles expressions régulières. □

5.3 Transformation des expressions régulières via des fonctions : version `dGREC`

Soit une spécification de schéma contenant des expressions de la forme $F : a[E]$. La version `dGREC` (direct `GREC`) de notre approche pour l'évolution des schémas propose des expressions régulières candidates en travaillant directement sur les expressions régulières E, c'est-à-dire, sans passer par la construction de l'automate. Cette méthode de transformation des expressions régulières a été proposée dans [dHM07, dL07] et est fondée sur l'utilisation des fonctions définies dans la section 5.1.2.

Comme dans `GREC`, la suppression est un cas plus simple que celui de l'insertion. L'insertion d'un symbole dans un mot $w \in L(E)$ donne lieu à un nouveau mot $w' \notin L(E)$. L'algorithme *Evolution* (algorithme 1) calcule des nouvelles expressions régulières E' proches de E telles que $w' \in L(E')$ et $L(E) \subseteq L(E')$. Remarquer que nous connaissons la position de mise à jour dans w.

Définition 5.3.1 *: Evolution*(E, w, i_{upd}) est un ensemble d'expressions régulières (obtenues par l'évolution de l'expression E) reconnaissant le mot $w' \notin L(E)$ où w' est obtenu par l'insertion d'un symbole à la position i_{upd} de w (donc $w' \in L(E')$ et $E' \in$ *Evolution*(E, w, i_{upd})). L'ensemble *Evolution*(E, w, i_{upd}) est défini par l'algorithme 1.

Étant donnée une expression régulière E, un mot $w \in L(E)$ et une position d'insertion i_{upd} sur w (en

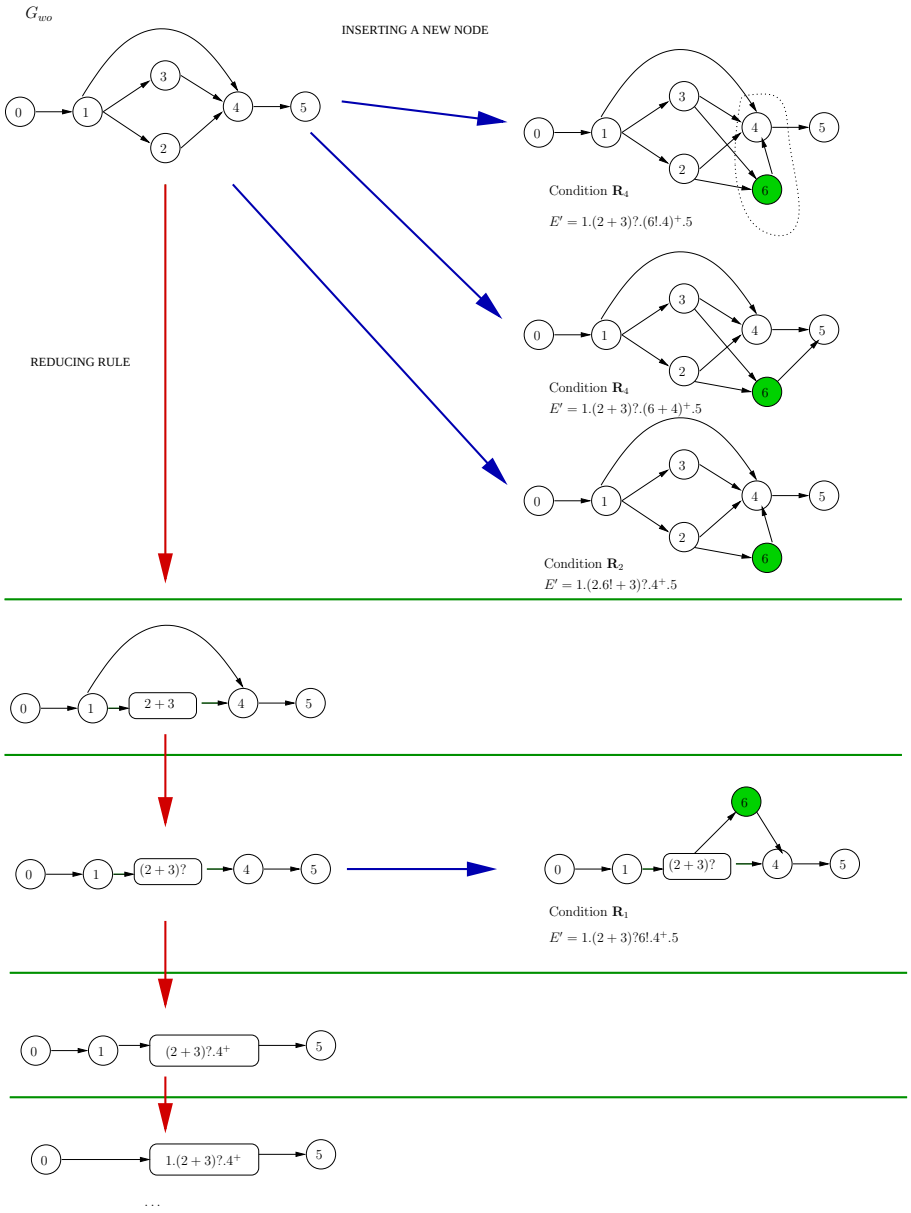


FIG. 5.5: Schéma de l'exécution de `GREC` pour $E = 1.(2+3)?.4^+.5$, $w = abd \in L(E)$ et $w' = abnd \notin L(E)$ et donc avec les positions $s_{nl} = 2$ et $s_{nr} = 4$. À gauche la réduction du graphe G_{wo} original. À droite des modifications effectuées pendant la procédure de réduction et quatre expressions régulières candidates.

Algorithm 1 *Evolution*(E, w, i_{upd})

function *Evolution*(E, w, i_{upd})
  var $R := \emptyset$ // Solutions calculated so far.
  var $C := \{p \mid p \in First(E) \text{ and } w_0 = symb(p)\}$ // Contient the current position in E.
  var $i := 0$ // Current symbol in w.
  var $C' := \emptyset$; // Next position in E.
  for $i := 0$ **to** $i_{upd} - 2$ **do**
    $C' := \{p' \mid p' \in Follow(E, p) \ \wedge \ w_{i+1} = symb(p') \ \wedge \ p \in C\}$
    $C := C'$
  end for
  $s_{nl} \in C$;
  $s_{nr} :=$ the position j in $Follow(E, s_{nl})$ such that $symb(j) = w_{i_{upd}}$
  $s_{new} := NewPos()$
  $R := R \cup \texttt{dGREC}(E, E, s_{nl}, s_{nr}, s_{new})$
  return R
end function

sachant que l'insertion donne lieu à $w' \notin L(E)$) l'algorithme *Evolution* exécute les étapes suivantes :

1. Une traversée, en parallèle du mot w et de l'expression régulière E en établissant une correspondance entre les positions de w et celles de E. La traversée sur w est faite en utilisant un entier i qui peut varier de 0 à i_{upd} ($0 < i_{upd} < |w|$) comme un "pointeur". La traversée sur E est faite en utilisant les fonctions présentées dans la section 5.1.2. Cette traversée permet de trouver la valeur de s_{nl}, c'est-à-dire, la position dans E ($s_{nl} \in Pos(E)$) qui correspond à la position $i_{upd} - 1$ du mot w.
2. La position de mise à jour sur w est i_{upd}. Autrement dit, dans le mot w' la position i_{upd} contient le symbole inséré. La position $s_{nr} \in Pos(E)$ est donc définie comme étant la position dans E qui correspond à la position i_{upd} de w (ou la postion $i_{upd} + 1$ de w').
3. Les positions s_{nl}, s_{nr} et une nouvelle position s_{new} sont utilisées pour calculer `dGREC`, l'ensemble des expressions régulières candidates E'. Remarquer que le rôle de la fonction *NewPos* est d'allouer une position nouvelle et non utilisée.

REMARQUES :
• Dans la méthode `dGREC` nous supposons l'existence d'un symbole spécial # au début et à la fin de chaque mot et de chaque expression régulière. Par exemple, l'expression $a.b^*$ est en effet vue comme l'expression $\#_0.a_1.b_2^*.\#_3$. Ainsi, $First(E)$ et $Last(E)$ sont toujours des ensembles unitaires. L'ensemble $First(E)$ contient la première position (0 dans l'exemple) alors que $Last(E)$ contient la dernière position (n) de E. De plus, $symb(First(E)) = symb(Last(E)) = \#$. De la même façon, un mot ab est vu comme $\#ab\#$. Grâce à ces symboles spéciaux l'algorithme *Evolution* reste simple, avec un traitement uniforme pour toutes les positions de mise à jour.
• Pour ne pas alourdir nos exemples (ici et dans [dHM07]) ces symboles spéciaux sont utilisés *seulement* si nécessaire pour la compréhension de l'exemple.
• L'algorithme *Evolution* (ainsi que le parcours sur un automate M_E dans l'approche `GREC`) définit la fonction map[3] qui associe une position i d'un mot v à une position p d'une expression régulière E ($v \in L(E)$). Ainsi, nous avons, pour $i = 0$

$$map(E, v, i) = p \text{ telle que } p \in First(E) \text{ et } v_0 = symb(p)$$

et pour $0 \leq i < |v| - 1$,

$$map(E, v, i + 1) = p' \text{ telle que } p' \in Follow(E, p) \ \wedge \ p \in map(E, v, i) \wedge \ v_{i+1} \in symb(p') \quad .$$

• Dans [dHM07], la procédure *Evolution* présentée considère l'insertion d'un symbole qui *n'apparaît pas* dans l'expression régulière originale. Ainsi, dans [dHM07], le but de la traversée est d'avancer jusqu'à rencontrer une position i (dans w') qui ne correspond à aucune position de E. Cela serait une autre manière de représenter le fait que nous connaissons la position de mise à jour sur w.

Le cœur de la fonction *Evolution* est le calcul de `dGREC` (définition 5.3.2). Pour faire évoluer une expression régulière, `dGREC` doit considérer différents types d'expressions régulières. Dans ce but les fonctions *evOr*, *evAnd*, *evClosure* et *evOpt* sont introduites (leur définition est présentée dans l'annexe B). Le calcul de `dGREC` est fait de gauche à droite : l'expression régulière E est partagée en sous expressions en suivant les opérateurs présents dans E. Les sous-expressions sont testées. Celles respectant certaines conditions sont transformées en des nouvelles sous expressions qui composeront les expressions régulières candidates.

Définition 5.3.2 *:* L'ensemble $\texttt{dGREC}(S, E, s_{nl}, s_{nr}, s_{new})$ contient les expressions régulières E' obtenues à partir de l'évolution de l'expression E. Chaque expression E' est obtenue à partir de E, par l'insertion d'une nouvelle position s_{new}, en accord avec les positions s_{nl} et s_{nr}. L'ensemble $\texttt{dGREC}(S, E, s_{nl}, s_{nr}, s_{new})$ est calculé par l'algorithme 2 où S est une expression régulière. □

Ci-dessous, nous illustrons le fonctionnement de la méthode `dGREC` en considérant l'expression $E = 1.(2+3)?.4^+.5$ de l'exemple 5.2.3. Les résultats de cet exemple sont présentés dans la figure 5.6 qui montre comment E est partagée et indique les opérateurs sur lesquels le partage se fait. Nous indiquons aussi les étapes où une transformation est activée et les expressions régulières obtenues.

Exemple 5.3.1 Comme dans l'exemple 5.2.3, nous considérons l'expression $E = 1.(2 + 3)?.4^+.5$ et les mots $w = abd \in L(E)$ et $w' = abnd \notin L(E)$. Nous avons donc $s_{nl} = 2$ et $s_{nr} = 4$. Le déroulement

[3]Remarquer que cette fonction est similaire à la fonction de transition de l'automate de Glushkov correspondant à une expression régulière E, définie dans[CZ00].

Algorithm 2 $\texttt{dGREC}(S, E, s_{nl}, s_{nr}, s_{new})$

$$
\begin{aligned}
\texttt{dGREC}(\emptyset, E, s_{nl}, s_{nr}, s_{new}) &= \emptyset \\
\texttt{dGREC}(\varepsilon, E, s_{nl}, s_{nr}, s_{new}) &= \emptyset \\
\texttt{dGREC}(\alpha_x, E, s_{nl}, s_{nr}, s_{new}) &= \emptyset \\
\texttt{dGREC}(F+G, E, s_{nl}, s_{nr}, s_{new}) &= evOr(F, G, E, s_{nl}, s_{nr}, s_{new}) \cup \\
&\quad \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new}) \cup \texttt{dGREC}(G, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC}(F.G, E, s_{nl}, s_{nr}, s_{new}) &= evAnd(F, G, E, s_{nl}, s_{nr}, s_{new}) \cup \\
&\quad \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new}) \cup \texttt{dGREC}(G, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC}(F^+, E, s_{nl}, s_{nr}, s_{new}) &= evClosure(F, E, s_{nl}, s_{nr}, s_{new}) \cup \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC}(F^*, E, s_{nl}, s_{nr}, s_{new}) &= evClosure(F, E, s_{nl}, s_{nr}, s_{new}) \cup \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new}) \\
\texttt{dGREC}(F?, E, s_{nl}, s_{nr}, s_{new}) &= evOpt(F, E, s_{nl}, s_{nr}, s_{new}) \cup \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new})
\end{aligned}
$$

de l'algorithme est illustré dans la figure 5.6. Remarquer que dans les appels récursifs, E est toujours l'expression d'origine alors que F et G représentent des sous expressions différentes à chaque appel.

Niveau 1 de figure 5.6 : L'algorithme 2 commence par partager E sur l'opération de concaténation. Plus précisément, nous avons :
$\texttt{dGREC}(F.G, E, s_{nl}, s_{nr}, s_{new}) = evAnd(F, G, E, s_{nl}, s_{nr}, s_{new}) \cup \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new}) \cup$
$\texttt{dGREC}(G, E, s_{nl}, s_{nr}, s_{new})$
où $F = 1$ et $G = (2+3)?.4^+.5$. Dans ce cas, aucune condition testée par $evAnd(F, G, E, s_{nl}, s_{nr}, s_{new})$ est satisfaite (annexe B). L'algorithme continue, notamment avec le calcul de $\texttt{dGREC}(G, E, s_{nl}, s_{nr}, s_{new})$.

Niveau 2 de figure 5.6 : Le calcul de $\texttt{dGREC}((2+3)?.4^+.5, E, s_{nl}, s_{nr}, s_{new})$ commence. Pour cela l'algorithme 2 indique que l'opération à faire est :
$\texttt{dGREC}(F.G, E, s_{nl}, s_{nr}, s_{new}) = evAnd(F, G, E, s_{nl}, s_{nr}, s_{new}) \cup \texttt{dGREC}(F, E, s_{nl}, s_{nr}, s_{new}) \cup$
$\texttt{dGREC}(G, E, s_{nl}, s_{nr}, s_{new})$
où $F = (2+3)?$ et $G = 4^+.5$ Dans ce cas, la première condition de $evAnd(F, G, E, s_{nl}, s_{nr}, s_{new})$ (annexe B, algorithme 4) est satisfaite et l'expression $E'_1 = 1.(2+3)?.6!.4^+.5$ est construite. En effet la condition teste si s_{nl} est une position finale de F et si s_{nr} est une position initiale de G pour pouvoir insérer la nouvelle position entre les deux.

Niveau 3 de figure 5.6 : Le raisonnement continue :
(i) Via $\texttt{dGREC}(F?, E, s_{nl}, s_{nr}, s_{new})$, nous avons $F = (2+3)$ du coté gauche. Les conditions de $evOpt(F, E, s_{nl}, s_{nr}, s_{new})$ ne sont pas vérifiées (annexe B, algorithme 6).
(ii) Via $\texttt{dGREC}(F.G, E, s_{nl}, s_{nr}, s_{new})$, nous avons $F = 4^+$ et $G = 5$ du coté droit. Des conditions de $evAnd(F, G, E, s_{nl}, s_{nr}, s_{new})$ ne sont pas vérifiées (annexe B, algorithme 4).

Niveau 4 de figure 5.6 : Finalement, les derniers appels récursifs nous donnent la situation suivante :
(i) Via $\texttt{dGREC}(F+G, E, s_{nl}, s_{nr}, s_{new})$ nous avons $F = 2$ et $G = 3$. La condition 1 de $evOr(F, G, E, s_{nl}, s_{nr}, s_{new})$ est satisfaite (annexe B, algorithme 7, case 1) et nous obtenons l'expression E'_2. La condition est satisfaite car $s_{nl} = 2$ est bien dans $Last(F)$ et $s_{nr} = 4$ est dans $Follow(E, 2)$. Cette condition teste si "l'erreur" est arrivée à la fin de l'expression F participant à un *or*. Si c'est le cas, la nouvelle position peut être ajoutée à la fin de F.
(ii) Via $\texttt{dGREC}(F^+, E, s_{nl}, s_{nr}, s_{new})$ nous avons $F = 4$ du coté droit. La condition 2 vérifiée dans $evClosure(F, E, s_{nl}, s_{nr}, s_{new})$ (annexe B, algorithme 7, case 2) permet la construction de E'_3 et E'_4. En effet, cette condition teste si "l'erreur" est arrivée avant l'entrée dans la boucle sur F (c'est-à-dire, dans l'orbite représentée par F^+). Dans ce cas, l'algorithme place la nouvelle position comme la première position de la boucle, c'est-à-dire, comme *First* de F ou comme une option à F.

Remarquer que dGREC trouve les mêmes solutions que GREC mais pas dans le même ordre. En effet, les conditions testées et les changements proposés sont les même dans les deux versions de la méthode. Néanmoins, dGREC fait un parcours de l'expression régulière de gauche à droite alors que GREC travaille à partir de l'orbite plus interne vers la plus externe. □

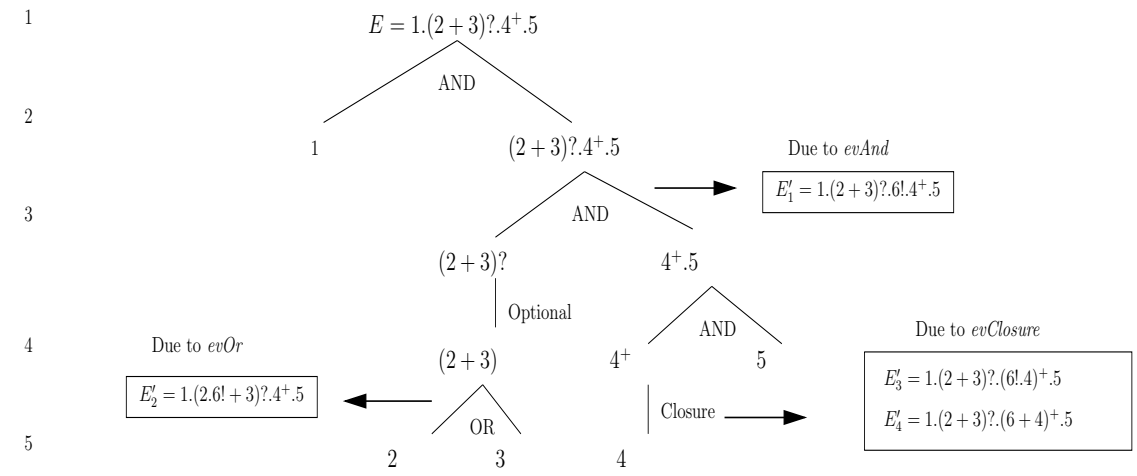


FIG. 5.6: Étapes de calcul `dGREC` pour $E = 1.(2+3)?.4^+.5$, $w = abd \in L(E)$ et $w' = abnd \notin L(E)$. Les positions $s_{nl} = 2$ et $s_{nr} = 4$ sont établies par *Evolution*.

Présentation de l'annexe B :
Nous avons présenté les idées générales du calcul des expressions régulières dans `GREC` et `dGREC` . Les détails concernant les transformations des expressions régulières sont montrés dans l'annexe B. Nous montrons les conditions testées et les changements effectués en faisant un parallèle entre les versions `dGREC` et `GREC`. Le but est de montrer que dans les deux approches les mêmes cas sont considérés et les mêmes solutions sont proposées. Remarquer que les deux méthodes ont été publiées indépendamment et c'est donc l'annexe B qui fait le parallèle entre les tests de chaque méthode, étape par étape.

Il est aussi important de remarquer que, dans ce mémoire, nous utilisons une version révisée de celle présentée dans [Dua05]. Dans la version révisée que nous proposons ici, la procédure de réduction reste la même (c'est-à-dire, celle introduite dans [CZ00] utilisant $Q^+(x)$ et $Q^-(x)$), mais les conditions à tester pour définir les solutions de `GREC` ne sont plus construites sur $Q^+(x)$ et $Q^-(x)$ (comme dans [Dua05]). En effet, ces conditions sont construites sur les ensembles $Foll(x)$ et $Prev(x)$ contenant les prédécesseurs et les successeurs d'un nœud x dans un graphe (et non dans un graphe sans cycle comme $Q^+(x)$ et $Q^-(x)$). Ces ensembles sont formellement définis dans l'annexe B.

5.4 Comparaisons de deux versions de la même approche : `GREC` et `dGREC`

La proposition suivante montre que les résultats obtenus dans les deux versions `GREC` et `dGREC` sont les mêmes.

Proposition 5.4.1 *Soit E une expression régulière et $L(E)$ son langage associé. Soit un mot $w[0:n] \in L(E)$. En supposant que i ($0 < i \leq n$) est une position de mise à jour*[4] *(insertion ou suppression) sur w, considérer : la suppression du symbole $w[i]$ qui donne comme résultat le mot $w_{del}[0:n-1]$; l'insertion du symbole $w[i]$ qui donne comme résultat le mot $w_{ins}[0:n+1]$. Soient $s_{nl} = map(E, w, i-1)$ et $s_{nr} = map(E, w, i)$ (pour une insertion) ou $s_{nr} = map(E, w, i+1)$ (pour une suppression). Soit $w' \notin L(E)$ un mot mis à jour (c'est-à-dire, w' est w_{del} ou w_{ins}). Soit s_{new} une position telle que $s_{new} \notin Pos(E)$.*

Soient les deux versions pour le calcul des transformations d'une expression régulière :

`dGREC`$(E, E, s_{nl}, s_{nr}, s_{new})$, calculé selon la définition 5.3.2, et

`GREC`$(G_{wo}, \mathcal{H}, s_{nl}, s_{nr}, s_{new})$, *calculé selon l'algorithme de la figure 5.4.*

Dans les deux versions, le résultat est un ensemble non vide d'expressions régulières E' telles que $L(E) \cup$

[4]Rappelons que les positions 0 et n sont les positions des marqueurs de début et de la fin du mot.

$\{w'\} \subseteq L(E')$. *De plus, la version* **dGREC** *est équivalente à la version* GREC. □

Preuve (*sketch*) : La preuve est fondée sur celles présentées dans [Dua05] où il a aussi été prouvé que les nouveaux graphes G_{new} engendrés par GREC sont des graphes de Glushkov et donc sont réductibles.

Dans GREC, les états s_{nl} et s_{nr} sont des nœuds du graphe G_{wo} ainsi que d'un graphe G_{new}. Ils sont donc, à un moment donné, concernés par la procédure de réduction. Dans dGREC les positions s_{nl} et s_{nr} sont dans l'expression régulière originale et sont aussi, à un moment donné, testées par une des fonctions participant au calcul de dGREC. La position s_{new} est insérée dans une sous-expression de E. En analysant chaque fonction de dGREC et chaque règle de GREC nous prouvons que, pour une insertion entre s_{nl} et s_{nr}, au moins une des conditions testées en GREC et dGREC est satisfaite et donc la solution est un ensemble non vide. La suppression consiste à rendre un nœud du graphe optionnel, c'est-à-dire, à ajouter seulement des arcs dans l'automate original. Un candidat est donc construit.

La correction de GREC et dGREC est démontrée par le fait que la position s_{new} est toujours incluse de façon optionnelle (et donc les mots originalement acceptés continuent à l'être) et en bien analysant la sous-expression dont elle doit faire partie (ce qui permet l'acceptation du mot w').

L'équivalence entre GREC et dGREC est prouvée en deux grandes étapes : (1) Chaque position x dans l'expression régulière E correspond à un état dans l'automate M_E. Cela est prouvé par la construction de M_E. Ensuite nous prouvons que, pour chaque état x dans M_E les ensembles $Foll(x)$ and $Prev(x)$ (annexe B) sont équivalents aux résultats obtenus par les fonctions $Follow(E, x)$ et $Previous(E, x)$, respectivement. (2) Il est possible de prouver ensuite que les tables concernant les règles $\mathbf{R}_1$, $\mathbf{R}_2$, $\mathbf{R}_3$ et $\mathbf{R}_4$ (figures B.1, B.2 B.3 et B.4) correspondent aux fonctions *evAnd*, *evOr*, *evOpt* et *evClosure*, respectivement et que les conditions testées et les solutions proposées sont les mêmes dans les deux cas. □

Dans les deux versions, dGREC et GREC, nous avons caractérisé les expressions régulières candidates via une notion de distance très simple. La distance entre deux expressions régulières E et E' est définie par $|\,|Pos(E)| - |Pos(E')|\,|$, c'est-à-dire, la valeur absolue de la différence entre le nombre de positions dans $\overline{E}$ et $\overline{E'}$. Il a été prouvé ([Dua05]) que toutes les expressions régulières candidates proposées par dGREC et GREC ont une distance 1 de l'expression régulière originale E.

Nous considérons maintenant la complexité des deux versions, GREC (calculée dans [Dua05]) et dGREC (calculée dans [dHM07]).

1. Le temps d'exécution de GREC est $O(n^2 + (n_j^2 \times k))$ où k est le nombre de graphes à réduire et n_j est le nombre de nœuds de chaque graphe j. En effet, chaque graphe modifié est transformé en expression régulière en temps $O(n_j^2)$ et GREC propose $k \geq 1$ graphes. La procédure de réduction du graphe original coûte $O(n^2)$.
2. Étant donnée une expression régulière E, le temps d'exécution de dGREC est $O(k \times (n+m)^2)$ où k est le nombre de fois que la fonction *Update* est utilisée, n est le nombre de positions dans E et m le nombre d'opérateurs de E. En effet, le calcul de l'ensemble dGREC pour une expression régulière E est fait en $(n+m)$ étapes récursives. Pour calculer les expressions candidates, nous avons besoin de la fonction *Update* qui est appelée k fois. Ainsi, la complexité est $O(k \times (n+m) \times$ *complexité de Update*$)$.

   La fonction *Update* cherche une sous expressions dans E pour la remplacer par une nouvelle sous expression. *Update* peut être implémentée de façon récursive et, dans ce cas, sa complexité est $O(n+m)$ en temps. Une optimisation de l'implémentation de *Update* est envisageable via l'utilisation d'un index pour les positions de l'expression régulière. Dans ce cas, la complexité de la fonction qui calcule l'ensemble dGREC doit être proche de $O(k \times (n+m))$.

À partir de la discussion ci-dessus, GREC et dGREC peuvent être considérées comme deux versions d'une même approche, donnant les mêmes résultats et ayant pratiquement la même complexité. Néanmoins, certaines différences peuvent renforcer les avantages d'une version par rapport à l'autre.

- dGREC semble être plus simple à comprendre et plus intuitif que GREC.
- La différence principale est, sans doute, l'utilisation ou non des automates d'états finis. La version GREC impose la construction d'un automate de Glushkov M_E pour chaque expression régulière E qui apparaît dans un schéma XML donné. Cette opération coûte, pour chaque E, $O(n^2)$ où n est le nombre des symboles dans E. dGREC n'utilise pas les automate de Glushkov mais des opérations sur les expressions régulières E dont la complexité est $O(n)$.
- GREC reste la version à choisir pour l'évolution des schémas lorsque les validateurs sont des automates

d'arbres, implémentés via l'API DOM (comme dans [BDHL03, Dua05]) puisque, dans ce cas, les automates d'états finis sont déjà utilisés dans la procédure de validation.
– dGREC est très facilement étendu pour traiter le problème de l'évolution incrémentale d'un validateur datalog (voir section 5.5). Il semble donc être la version à choisir pour l'évolution des schémas lorsque les validateurs sont représentés par un programme logique (comme datalog).

5.5 Évolution du validateur datalog : version dGREC^{Dat}

Dans les sections précédentes, nous avons considéré la description du schéma via des expressions régulières. Dans cette section, nous nous plaçons dans le même contexte de GREC et dGREC, mais nous utilisons un programme datalog pour décrire un schéma XML : chaque expression régulière est donc traduite dans un ensemble de règles datalog. L'union des règles correspondant à chaque expression régulière est le programme datalog correspondant au schéma XML.

Nous supposons ensuite des mises à jour sur les positions d'un arbre XML. Comme dans GREC et dGREC, nous considérons une seule mise à jour (insertion ou suppression) sur une position de l'arbre XML. Si la mise à jour viole le schéma (c'est-à-dire, viole l'expression régulière associée au père de la position de mise à jour), nous ajoutons des règles datalog au programme original pour prendre en compte l'évolution du schéma (c'est-à-dire, l'évolution de l'expression régulière). Ainsi, l'évolution de E vers E' est présentée comme l'évolution du programme datalog $\mathcal{P}$ vers $\mathcal{P}'$. Il ne s'agit pas de calculer E' pour ensuite la traduire en règles datalog, mais d'ajouter à l'ensemble de règles datalog représentant E des nouvelles règles représentant sa transformation en E'.

Cette méthode représente ainsi une réécriture de la méthode dGREC en termes de programme datalog. Remarquer que le programme datalog est aussi un validateur de contraintes de schéma.

Dans la suite, la section 5.5.1 présente un algorithme pour traduire un schéma XML en programme datalog. Ensuite, dans la section 5.5.2 nous présentons dGREC^{Dat}, une version étendue de dGREC capable de faire l'évolution incrémentale de notre validateur datalog.

5.5.1 Construction d'un validateur comme un programme Datalog

L'algorithme de traduction utilise les fonctions de la section 5.1.2 et s'inspire de la preuve du théorème 2.1.2 (qui indique aussi les prédicats extensionnels utilisés).

Soit $F : a[E]$ une spécification de schéma. Cette spécification impose une contrainte à un nœud étiqueté par a, plus précisément, elle impose une contrainte, exprimée par l'expression régulière E, aux enfants du nœud en question. Étant donné un nœud (ou une position) x dans un arbre XML, pour vérifier s'il respect la contrainte, nous devons parcourir tous ses fils de gauche à droite, en vérifiant si leurs schémas sont valides. L'originalité de notre approche, introduit dans [dHM07], est d'utiliser les fonctions de la section 5.1.2 pour effectuer ce parcours.

REMARQUE : Par abus de notation, dans la définition suivante, pour une position i, nous notons :
(i) $P_{symb(i)}$ pour indiquer l'étiquette associée à la position,
(ii) $sch_{symb(i)}$ pour indiquer le schéma associé aux fils de cette position et
(iii) $\mathbf{Trad}(symb(i))$ pour indiquer l'expression régulière que nous voulons traduire.
Néanmoins, nous utilisons un terminal comme indice pour $P_{symb(i)}$ et un non terminal comme indice pour $sch_{symb(i)}$ et $\mathbf{Trad}(symb(i))$. Par exemple, soit $A : a[B^*.C]$, nous notons P_a pour indiquer l'étiquette et sch_A et $\mathbf{Trad}(A)$ pour indiquer le(s) schéma(s) associés aux nœuds d'étiquette a.

Définition 5.5.1 *:* Soit $F : a[E]$ une spécification de schéma où F est un non terminal, a est un symbole (terminal) dans Σ et E est une expression régulière sur les non terminaux. La traduction de F en programme Datalog est faite selon la fonction récursive **Trad** définie ci-dessous :

• For $E = \epsilon$:
$\mathbf{Trad}(F) =$ "$sch_F(x) \leftarrow P_a(x), Leaf(x)$"

• For $E \neq \epsilon$:
$\mathbf{Trad}(F) =$

"$sch_F(x) \leftarrow P_a(x), children_E(x)$"
If $(E = G^*)$ or $(E = G?)$
"$sch_F(x) \leftarrow P_a(x), Leaf(x)$"
For all $i \in Last(E)$:
"$children_E(x) \leftarrow LC(x,y), P_{symb(i)}(y), sch_{symb(i)}(y), posRE_{E,i}(y)$"
Trad$(symb(i))$
For all $i \in Pos(E)$; for $j \in Follow(E,i)$:
If $i \in First(E)$: "$posRE_{E,i}(y) \leftarrow FC(x,y), P_{symb(i)}(y), sch_{symb(i)}(y)$" ; **Trad**$(symb(i))$
"$posRE_{E,j}(y) \leftarrow posRE_{E,i}(x), NS(x,y), P_{symb(j)}(y), sch_{symb(j)}(y)$"
Trad$(symb(j))$ □

Tout d'abord, la fonction **Trad** partage la traduction en deux groupes à savoir, le cas où le nœud étiqueté a est une feuille $(E = \epsilon)$ et le cas contraire. Le premier cas est trivial, la traduction correspond donc à une seule règle datalog. Le deuxième cas $(E \neq \epsilon)$ est le cœur de la fonction de traduction et correspond à un programme datalog construit selon les étapes suivantes :

1. La première règle $sch_F(x) \leftarrow P_a(x), children_E(x)$ établie l'idée générale de la validation d'un nœud x par rapport à la contrainte $a[E]$, à savoir
   - le nœud x est étiqueté a et
   - ses enfants respectent leurs contraintes de schéma.

   Quand l'expression régulière est de la forme $(E = G^*)$ ou $(E = G?)$ le nœud x peut ne pas avoir d'enfants. La règle $sch_{a[E]}(x) \leftarrow P_a(x), Leaf(x)$ est donc ajoutée au programme datalog.
2. Le reste de l'algorithme traite de la validation des enfants de x.

   (a) La règle $children_E(x) \leftarrow LC(x,y), P_{symb(i)}(y), sch_{symb(i)}(y), posRE_{E,i}(y)$ vérifie si y, le dernier fils de x, respecte les conditions suivantes :
   - y est associé au label α tel qu' il existe $i \in Last(E)$ et $symb(i) = \alpha$;
   - le schéma de y (étant donné par le schéma de spécification associé à $symb(i)$) est respecté par ses enfants et
   - y est est la dernière position atteinte à partir de ses frères à gauche, en parcourant en parallèle le mot des fils de x et l'expression régulière E.

   (b) La validation de schéma doit continuer par la vérification du schéma des fils de x de gauche à droite jusqu'à atteindre le dernier fils. Pour cela, il faut établir un parcours du nœud[5] $u.0$ au nœud $u.n$ en vérifiant le schéma de chaque nœud. Comme nous supposons que le nœud x est associé à la contrainte $a[E]$, pour effectuer ce parcours, il est nécessaire d'avoir des règles datalog capables d'associer un enfant de x sur l'arbre XML au symbole auquel il correspond dans l'expression régulière E. Pour cela, pour chaque $i \in Pos(E)$ et pour chaque $j \in Follow(E,i)$; nous définissons les relations $posRE_{E,j}$. Une relation $posRE_{E,j}$ fait l'association entre une position $y \in D$ de l'arbre XML à une position j dans E.
   - La règle $posRE_{E,i}(y) \leftarrow FC(x,y), P_{symb(i)}(y), sch_{symb(i)}(y)$ initialise les relations $posRE_{E,i}$ pour chaque $i \in First(E)$: chacune peut contenir le premier fils de x (pourvu qu'il respecte les contraintes de schéma).
   - La règle $posRE_{E,j}(y) \leftarrow posRE_{E,i}(y_1), NS(y_1,y), P_{symb(j)}(y), sch_{symb(j)}(y)$ ajoute aux relations $posRE_{E,j}$ les positions y représentant chaque fils de x qui respecte toutes les conditions suivantes :
     - y_1, le frère immédiatement à gauche de y, est déjà dans $posRE_{E,i}$ (for $i \leq j$) et son schéma n'est pas violé.
     - y est associé au label α tel que il existe $j \in Follow(E,i)$ et $symb(j) = \alpha$ et son schéma est respecté.

     De manière intuitive, nous pouvons dire que les relations $posRE_{E,j}$ nous permettent d'effectuer des parcours en parallèle : un sur les enfants de x et l'autre sur l'expression régulière associée.

Exemple 5.5.1 Soit un extrait de l'exemple complet présenté dans [dHM07]. Supposons un schéma où $A : a[B^*.C]$. En utilisant les fonctions de la section 5.1.2, nous avons : $First(B_0^*.C_1) = \{0,1\}$; $Last(B_0^*.C_1) = \{1\}$ et $Follow(B_0^*.C_1, 0) = \{0,1\}$; $Follow\ (B_0^*.C_1, 1) = \emptyset$.

[5] Supposer un nœud x correspondant à la position u et ayant $n+1$ enfants.

Soit $E = B^*.C$. Alors, $\mathbf{Trad}(A)$ où $A : a[E]$ donne comme résultat :

$sch_A(x) \leftarrow P_a(x), children_E(x)$
$children_E(x) \leftarrow LC(x,y), P_c(y), sch_C(y), posRE_{E,1}(y)$
$posRE_{E,0}(y) \leftarrow FC(x,y), P_b(y), sch_B(y)$
$posRE_{E,1}(y) \leftarrow FC(x,y), P_c(y), sch_C(y)$
$posRE_{E,0}(y) \leftarrow posRE_{E,0}(x), NS(x,y), P_b(y), sch_B(y)$
$posRE_{E,1}(y) \leftarrow posRE_{E,0}(x), NS(x,y), P_c(y), sch_C(y)$
$\mathbf{Trad}(B)\,;\, \mathbf{Trad}(C)\,;$ □

5.5.2 Évolution du programme Datalog selon l'évolution du schéma

Dans cette section nous considérons $\texttt{dGREC}^{Dat}$, une variation de dGREC qui au lieu de changer une expression régulière, change un programme datalog $\mathcal{P}$ correspondant à un schéma XML. À partir de la mise à jour sur un fils d'une position u de l'arbre XML, dGREC construit des expressions E' alors $\texttt{dGREC}^{Dat}$, ajoute des règles à $\mathcal{P}$ (pour exprimer ainsi le changement de E vers E'). L'évolution de $\mathcal{P}$ est donc incrémentale : il ne s'agit pas d'obtenir E' et de la traduire, mais de déterminer directement les règles datalog à ajouter dans $\mathcal{P}$.

Le contexte considéré est le même de dGREC. Soit $w' \notin L(E)$ le mot résultat d'une mise à jour sur $w \in L(E)$. Les positions s_{nl}, s_{nr} et s_{new} sont calculées comme dans $\texttt{dGREC}^{Dat}$. Soit $\mathcal{P}$ le programme datalog qui représente le schéma XML. L'ensemble $\texttt{dGREC}^{Dat}$ contient les règles datalog à ajouter dans $\mathcal{P}$ pour exprimer l'évolution proposé par dGREC de E en E'.

Définition 5.5.2 Soient E une expression régulière, $F : a[E]$ la spécification d'un schéma et $\mathcal{C} = \mathbf{Trad}(F)$. L'ensemble des règles Datalog $\texttt{dGREC}^{Dat}(\mathcal{C}, S, E, s_{nl}, s_{nr}, s_{new})$ est obtenu par l'évolution de l'ensemble $\mathcal{C}$, en accord avec l'évolution de l'expression régulière E, par l'insertion de s_{new} dans E. L'évolution est guidée par les positions s_{nl} et s_{nr}. S est une expression régulière.

Ainsi si $E' \in \texttt{dGREC}(S, E, s_{nl}, s_{nr}, s_{new})$, alors
$\texttt{dGREC}^{Dat}(\mathcal{C}, S, E, s_{nl}, s_{nr}, s_{new}) = \quad \{\mathcal{C}' \mid (\mathcal{C} \cup \mathcal{C}' \cup \mathbf{Trad}(symb(s_{new}))) = \mathbf{Trad}(a[E'])\}$.
L'ensemble $\texttt{dGREC}^{Dat}$ est calculé par l'algorithme 3. □

Algorithm 3 $\texttt{dGREC}^{Dat}(\mathcal{C}, S, E, s_{nl}, s_{nr}, s_{new})$

$\texttt{dGREC}^{Dat}(\mathcal{C}, \emptyset, E, s_{nl}, s_{nr}, s_{new}) = \emptyset$
$\texttt{dGREC}^{Dat}(\mathcal{C}, \varepsilon, E, s_{nl}, s_{nr}, s_{new}) = \emptyset$
$\texttt{dGREC}^{Dat}(\mathcal{C}, \alpha_x, E, s_{nl}, s_{nr}, s_{new}) = \emptyset$
$\texttt{dGREC}^{Dat}(\mathcal{C}, F+G, E, s_{nl}, s_{nr}, s_{new}) = evOr^{Dat}(\mathcal{C}, F, G, E, s_{nl}, s_{nr}, s_{new}) \cup$
$\quad \texttt{dGREC}^{Dat}(\mathcal{C}, F, E, s_{nl}, s_{nr}, s_{new}) \cup \texttt{dGREC}^{Dat}(\mathcal{C}, G, E, s_{nl}, s_{nr}, s_{new})$
$\texttt{dGREC}^{Dat}(\mathcal{C}, F.G, E, s_{nl}, s_{nr}, s_{new}) = evAnd^{Dat}(\mathcal{C}, F, G, E, s_{nl}, s_{nr}, s_{new}) \cup$
$\quad \texttt{dGREC}^{Dat}(\mathcal{C}, F, E, s_{nl}, s_{nr}, s_{new}) \cup \texttt{dGREC}^{Dat}(\mathcal{C}, G, E, s_{nl}, s_{nr}, s_{new})$
$\texttt{dGREC}^{Dat}(\mathcal{C}, F^+, E, s_{nl}, s_{nr}, s_{new}) = evClosure^{Dat}(\mathcal{C}, F, E, s_{nl}, s_{nr}, s_{new}) \cup$
$\quad \texttt{dGREC}^{Dat}(\mathcal{C}, F, E, s_{nl}, s_{nr}, s_{new})$
$\texttt{dGREC}^{Dat}(\mathcal{C}, F^*, E, s_{nl}, s_{nr}, s_{new}) = evClosure^{Dat}(\mathcal{C}, F, E, s_{nl}, s_{nr}, s_{new}) \cup$
$\quad \texttt{dGREC}^{Dat}(\mathcal{C}, F, E, s_{nl}, s_{nr}, s_{new})$
$\texttt{dGREC}^{Dat}(\mathcal{C}, F?, E, s_{nl}, s_{nr}, s_{new}) = evOpt^{Dat}(\mathcal{C}, F, E, s_{nl}, s_{nr}, s_{new}) \cup$
$\quad \texttt{dGREC}^{Dat}(\mathcal{C}, F, E, s_{nl}, s_{nr}, s_{new})$

Comme $\mathcal{P} \subseteq \mathcal{P}'$, tous les arbres XML valides par rapport à $\mathcal{P}$ restent valides par rapport à $\mathcal{P}'$. Soit $\mathcal{C}$ l'ensemble des règles obtenues par $\mathbf{Trad}(F)$ où $F : a[E]$. La définition de $\texttt{dGREC}^{Dat}$ est similaire à celle de dGREC (section 2), mais au lieu de considérer les changements sur les expressions régulières E, nous prenons en compte les changements de $\mathcal{C}$, dû à l'évolution de E. Ainsi, $\texttt{dGREC}^{Dat}$ est l'ensemble des règles Datalog obtenues par l'évolution d'un ensemble (de règles datalog) $\mathcal{C}$.

Le calcul de $\texttt{dGREC}^{Dat}(S, E, s_{nl}, s_{nr}, s_{new})$ est similaire au calcul de dGREC. Nous remplaçons dGREC par $\texttt{dGREC}^{Dat}$ (en adaptant les paramètres) et *evOr*, *evAnd*, *evOpt* et *evClosure* par $evOr^{Dat}$, $evAnd^{Dat}$, $evOpt^{Dat}$ et $evClosure^{Dat}$, respectivement. Ces nouvelles fonctions définissent les règles à ajouter.

Les règles datalog à ajouter
Comme E représente le schéma qui doit être respecté par les enfants d'un nœud p, l'insertion de s_{new} correspond à l'insertion d'un nouveau fils de p. Cette insertion implique l'addition des nouvelles règles datalog dans $\mathcal{P}$. La définition de ces nouvelles règles est guidée par les positions s_{nl} et s_{nr}. Nous rappelons que s_{nl} et s_{nr} sont des positions correspondant à des symboles existant dans E, alors que s_{new} peut correspondre à un nouveau symbole.

Les nouvelles règles doivent, tout d'abord, prendre en compte que : (a) s_{nl} peut avoir s_{new} comme frère à droite ; (b) s_{nr} peut avoir s_{new} comme frère à gauche ; (c) s_{new} peut avoir un autre s_{new} comme frère à gauche. Ainsi, le programme $\mathcal{P}$ original est transformé en $\mathcal{P}_1$ tel que $\mathcal{P}_1 = \mathcal{P} \cup \mathcal{P}_{stand}$ où
$\mathcal{P}_{stand} = \{\ posRE_{E,s_{new}}(y) \leftarrow posRE_{E,s_{nl}}(x), NS(x,y), P_{symb(s_{new})}(y), sch_{symb(s_{new})}(y)$
$posRE_{E,s_{new}}(y) \leftarrow posRE_{E,s_{new}}(x), NS(x,y), P_{symb(s_{new})}(y), sch_{symb(s_{new})}(y)$
$posRE_{E,s_{nr}}(y) \leftarrow posRE_{E,s_{new}}(x), NS(x,y), P_{symb(s_{nr})}(y), sch_{symb(s_{nr})}(y)\ \}$

Remarquer qu'il n'est pas nécessaire d'ajouter des règles pour définir $posRE_{E,s_{nl}}$ et $posRE_{E,s_{nr}}$ puisqu'elles existent déjà dans $\mathcal{P}$.

Ensuite, nous définissions les ensembles $\mathcal{P}_{first}$, $\mathcal{P}_{last}$ et $\mathcal{P}_{loop}$:

- Si s_{new} peut être le premier fils de p, alors
  $\mathcal{P}_{first} = \{posRE_{E,s_{new}}(y) \leftarrow FC(x,y), P_{symb(s_{new})}(y),\ sch_{symb(s_{new})}(y)\}$.
  Sinon, $\mathcal{P}_{first} = \emptyset$.
- Si s_{new} peut être le dernier fils de p, alors
  $\mathcal{P}_{last} = \{children_E(x) \leftarrow LC(x,y), P_{symb(s_{new})}(y), sch_{symb(s_{new})}(y),\ posRE_{E,s_{new}}(y)\}$.
  Sinon, $\mathcal{P}_{last} = \emptyset$.
- Si s_{new} peut être la première ou la dernière position d'une boucle E^* ou E^+, alors
  $\mathcal{P}_{loop} = \{posRE_{E,j}(y) \leftarrow posRE_{E,s_{new}}(x), NS(x,y), P_{symb(j)}(y), sch_{symb(j)}(y) \mid j \in First(E)\} \cup$
  $\{posRE_{E,s_{new}}(y) \leftarrow posRE_{E,i}(x), NS(x,y), P_{symb(s_{new})}(y), sch_{symb(s_{new})}(y) \mid i \in Last(E)\}$
  Sinon, $\mathcal{P}_{loop} = \emptyset$.

Par conséquent, $\mathcal{P}_2 = \mathcal{P}_1 \cup \mathcal{P}_{first} \cup \mathcal{P}_{last} \cup \mathcal{P}_{loop}$.

Finalement, les règles décrivant le schéma de $symb(s_{new})$ doivent être ajoutées au programme datalog. Ainsi pour obtenir $\mathcal{P}'$, nous ajoutons à $\mathcal{P}_2$ les règles obtenues par le calcul de $\mathbf{Trad}(symb(s_{new}))$.

L'ensemble $\texttt{dGREC}^{Dat}(\mathcal{C}, S, E, s_{nl}, s_{nr}, s_{new})$ est calculé par l'algorithme 3. Notre approche permet la maintenance incrémentale d'un validateur datalog $\mathcal{P}$, construit à partir d'un schéma XML donné. Chaque $\mathcal{P}'$ dans $\texttt{dGREC}^{Dat}$ correspond à l'évolution de $\mathcal{P}$ dû à une expression E', obtenue par dGREC comme évolution possible de E. Dans ce cadre, chaque nouveau programme $\mathcal{P}'$ est définit comme $\mathcal{P}' = \mathcal{P} \cup \mathcal{C}' \cup \mathbf{Trad}(s_{new})$ où $\mathcal{C}' \in \texttt{dGREC}^{Dat}(\mathcal{C}, S, E, s_{nl}, s_{nr}, s_{new})$. Il est important de remarquer que, dans la version datalog, l'évolution de E vers E' via l'introduction d'une nouvelle position dans l'expression régulière d'origine est traduite non seulement par les règles explicitées ci-dessus, mais aussi par des règles définissant le schéma associé au symbole inséré. Ainsi, quand nous ajoutons les règles de $\mathbf{Trad}(symb(snew))$, nous ajoutons, en effet les règles définissant le sous arbre à insérer dans l'arbre XML.

Dans $\texttt{dGREC}^{Dat}(S, E, s_{nl}, s_{nr}, s_{new})$, les fonctions $evOr^{Dat}$, $evAnd^{Dat}$, $evOpt^{Dat}$ et $evClosure^{Dat}$ sont définies de façon similaire à leurs versions sur les expressions régulières, en ajoutant des tests. Plus précisément, dans $evOr^{Dat}$, $evAnd^{Dat}$, $evOpt^{Dat}$ et $evClosure^{Dat}$ nous testons : (a) si les positions s_{nl} et s_{nr} vérifient les conditions capables de déclencher des changements (comme dans la version sur les expressions régulières) et (b) si s_{new} peut être premier ou dernier fils ou le début ou fin d'une boucle. Remarquer que les tests concernant $\mathcal{P}_{loop}$ sont effectués seulement dans $evClosure^{Dat}$.

Exemple 5.5.2 Soit $\mathcal{P}$ le validateur datalog contenant (entre autres) les règles de l'exemple 5.5.1. Supposons un arbre XML où la racine est associée au label a et où la concaténation de ses enfants donne le mot *bbc*. Une mise à jour sur cet arbre transforme le mot *bbc* dans le mot *bbmc* en violant la contrainte de schéma. Dans ce cas, notre expression régulière est $E = B_0^*.C_1$; $s_{nl} = 0$; $s_{nr} = 1$ et $s_{new} = 2$. La méthode dGREC obtient les expressions régulières $E_1 = B_0^*.M_2!.C_1$, $E_2 = (B_0.M_2!)^*.C_1$, et $E_3 = (M_2! + B_0)^*.C_1$.

Remarquer que, dans ce cas, il s'agit de l'insertion d'un sous arbre dont la racine a l'étiquette m. Les fils de cette racine doivent respecter le schéma défini par M qui est supposé déjà connu (néanmoins, si $\mathbf{Trad}(s_{new}) = \mathbf{Trad}(M)$ n'est pas connu, il suffit d'appliquer la fonction **Trad**, définition 5.5.1, pour le calculer). En plus des règles de $\mathbf{Trad}(M)$, nous devons ajouter les règles correspondant à l'évolution de

l'expression E. En effet, $\texttt{dGREC}^{Dat}$ obtient les programmes $\mathcal{P}_1$ $\mathcal{P}_2$ et $\mathcal{P}_3$ correspondant à E_1 ; E_2 ; et E_3 ; respectivement. Le programme $\mathcal{P}_1$ est obtenu par $evAnd^{Dat}$ (case 1) alors que $\mathcal{P}_2$ et $\mathcal{P}_3$ sont obtenus par $evClosure^{Dat}$ (case 3).

Pour obtenir $\mathcal{P}_3$, les règles suivantes sont à ajouter :
1) $posRE_{E,2}(y) \leftarrow posRE_{E,0}(x), NS(x,y), P_m(y), sch_M(y),$;
2) $posRE_{E,2}(y) \leftarrow posRE_{E,2}(x), NS(x,y), P_m(y), sch_M(y)$;
3) $posRE_{E,1}(y) \leftarrow posRE_{E,2}(x), NS(x,y), P_c(y), sch_C(y)$;
4) $posRE_{E,2}(y) \leftarrow FC(x,y), P_m(y),\ sch_M(y)$;
5) $posRE_{E,0}(y) \leftarrow posRE_{E,2}(x), NS(x,y), P_b(y), P_B(y)$;

Remarquer que $\mathcal{P}_{stand}$ correspond aux règles de 1-3 ; $\mathcal{P}_{first}$ correspond à la règle 4 alors que dans P_{loop} la seule règle qui n'est pas incluse dans $\mathcal{P}_{stand}$ est la règle 5. Les autres programmes sont des sous ensemble de $\mathcal{P}_3$. Ainsi $\mathcal{P}_1$ ne contient pas la règle 5 alors que $\mathcal{P}_2$ ne contient pas la règle 4. □

Proposition 5.5.1 *Soit E' une expression régulière dans l'ensemble* ***dGREC*** *obtenue à partir d'une expression E et des positions s_{nl}, s_{nr}, s_{new} (proposition 5.4.1). Le calcul de $\texttt{dGREC}^{Dat}$ (définition 5.5.2) est correct, c'est-à-dire, étant donné un ensemble $\mathcal{C}' \in \texttt{dGREC}^{Dat}(\mathcal{C}, E, E, s_{nl}, s_{nr}, s_{new})$, l'ensemble $(\mathcal{C} \cup \mathcal{C}' \cup \textbf{\textit{Trad}}(symb(s_{new})))$, où $\mathcal{C}$ est le résultat de $\textbf{\textit{Trad}}(F)$ (avec $F : a[E]$), est égal au résultat obtenu par $\textbf{\textit{Trad}}(F')$ (avec $F' : (a[E'])$.* □

Preuve (*sketch*) : Voir [dHM07]

Proposition 5.5.2 *Soit $\mathcal{P}$ un programme Datalog contenant $\textbf{\textit{Trad}}(F)$ (où $F : a[E]$) et ayant le prédicat "valid" comme réponse souhaitée (section 5.5.1) . Soit T un arbre tel que $\mathcal{P}(T)$ est "valid". Supposons une mise à jour (insertion ou suppression) sur les enfants d'un nœud p de T qui résulte en T', tel que $\mathcal{P}(T')$ n'est plus "valid". Supposons que cette mise à jour déclenche une évolution de schéma (avec s_{nl}, s_{nr}, s_{new} comme dans la proposition 5.4.1).*

Soit $\mathcal{P}'$ un programme datalog obtenu par l'ajout de $\mathcal{C}' \in \texttt{dGREC}^{Dat}(\mathcal{C}, E, E, s_{nl}, s_{nr}, s_{new})$ et du résultat de $\textbf{\textit{Trad}}(symb(s_{new}))$ à $\mathcal{P}$. Pour tout arbre T tel que $\mathcal{P}(T)$ est évalué comme "valid"; $\mathcal{P}'(T)$ est aussi évalué comme "valid". De plus, $\mathcal{P}'(T')$ est évalué comme "valid". □

Preuve (*sketch*) : Voir [dHM07]

Une caractéristique très intéressante de $\texttt{dGREC}^{Dat}$ est la possibilité de manipuler/faire évoluer le type du document XML intégralement et non seulement une partie (via les expressions régulières).

5.6 Extensions et implémentations

Dans notre approche, l'évolution de schéma est déclenchée à partir d'une unique mise à jour sur un document XML. Une extension naturelle de cette méthode consiste à prendre en compte une liste de mises à jour au lieu d'une seule. Cette extension à été considérée dans la cadre de la version GREC ([Dua05, BDHM07]) pour donner naissance à GREC-e. Dans cette extension, le mot w' est le résultat d'une suite de mises à jour sur w. Dans ce mémoire nous ne présentons pas les détails de cette extension, mais simplement une rapide discussion sur ce point. Pour plus des détails, voir [Dua05, BDHM07].

L'algorithme qui calcule l'ensemble d'expressions régulières candidates pour GREC-e est similaire à l'algorithme qui calcule les expressions pour GREC. Néanmoins, l'extension GREC-e doit prendre en compte plusieurs n-uplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$, un pour chaque mise à jour sur w. Ainsi, dans l'extension GREC-e la boucle (lignes $07-08$) de l'algorithme de la figure 5.4 est modifiée par l'ajout d'un appel récursif. La fonction *LookForGraphAlternative* propose des graphes candidats G_1 en prenant en compte un n-uplet $\langle s_{nl}, s_{nr}, s_{new}\rangle$. Ensuite, sur chaque graphe candidat G_1, GREC-e est appelé dans le but construire un candidat, à partir de G_1, en prenant en compte les autres n-uplets (pas encore traités). En effet, pour chaque graphe candidat pour une insertion, des nouveaux graphes sont construits en considérant les autres n-uplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$ et ainsi de suite. Soit k le nombre de graphes candidats en appliquant une fois les modifications proposées par GREC-e. Dans le pire de cas, GREC-e construit k^n candidats où n est le nombre de n-uplets $\langle s_{nl}, s_{nr}, s_{new}\rangle$. Comme les insertions sur une même position sont considérées dans une même étape, la valeur n est atteinte seulement quand les positions de mise à jour sont toutes distinctes.

En rappelant que la complexité de la procédure de réduction est $O(m^2)$ (section 5.4), où m est le

nombre des nœuds d'un graphe candidat donné, la complexité de GREC-e est $O(k^n \times m^2)$. Malgré ce résultat décourageant, les tests effectués ont montré la viabilité de l'utilisation de GREC-e dans le contexte d'évolution de schéma[6]. Les résultats de nos expérimentations ont été résumés dans [BDHM07].

Il est intéressant de remarquer que la relation contenant les n-uplets $\langle s_{nl}, s_{nr}, s_{new} \rangle$ est construite pendant un pré-traitement qui consiste à comparer w et w' ([Dua05, BDHM07]). Une autre relation, appelée $STrans$ est aussi construite dans l'étape de pré-traitement. Elle contient des contraintes supplémentaires qui imposent un ordre sur les nouveaux états s_{new} et doivent être prises en compte dans la construction des graphes candidats.

Un point critique de notre méthode pour l'évolution des schéma (plus précisément des expressions régulières) est le nombre de candidats. Dans l'extension GREC-e ce problème est encore plus important. Une façon d'améliorer la situation est de proposer des choix à l'utilisateur de façon à diminuer le nombre de cas à exploiter.

La version dGREC n'a pas encore été étendue pour traiter plusieurs mises à jour mais cela est possible en suivant les mêmes idées appliquées dans GREC-e. L'implémentation de dGREC pour une seule mise à jour est détaillée dans [dL07] et présente de très bons résultats. Une optimisation de la fonction *Update* est à prévoir.

5.7 Travaux liés

La plupart des travaux sur l'évolution de schéma (ou type) propose des opérations pour le changement du schéma et, ensuite, différentes manières d'adapter les documents au nouveau type. L'état de l'art de la thèse [Dua05] discute certains de ces travaux. Les propositions dans [KLP02, SKR01] sont similaires dans leur démarche ayant trois étapes principales : (1) Établissement d'un ensemble d'opérations de transformation de type en prenant en compte des relations sémantiques entre les schémas. (2) Proposition d'un modèle qui leur permet de comparer le coût de l'application des opérations. (3) Présentation d'un algorithme, fondé sur le modèle de coût, qui calcule une suite d'opérations pour transformer un type source dans un type cible. La séquence d'opérations est utilisée pour engendrer les opérations de transformation des documents XML.

Dans [GMR05] un ensemble de primitives pour changer le schéma est proposé. Cet ensemble de primitives d'évolution est correct (appliqué sur un schéma cohérent[7] fournit un schéma cohérent) et complet (n'importe quel schéma peut être engendré à partir du schéma vide en appliquant les primitives). Ensuite, l'impact de l'évolution sur les documents originalement valides est analysé. L'idée de base est d'éviter la validation de tout le document, en identifiant les parties du schéma qui, dû fait des changements, demandent une revalidation. Cela est fait par l'examen de la primitive d'évolution et ses paramètres. Tout d'abord, les auteurs déterminent les primitives qui n'ont aucun impact sur la validité (*validity preserving primitives*) comme par exemple l'insertion d'un élément optionnel ou une extension de cardinalité. Ensuite, ils considèrent que le nouveau schéma Υ', obtenu à partir du schéma Υ via des primitives d'évolution, est représenté par un graphe (*type graph*). Un schéma est vu comme un type composé de sous types définissant des sous arbres. Soient τ et τ' des sous types de Υ et Υ', respectivement. Une procédure d'étiquetage associe un label à chaque nœud du graphe qui correspond à un type en suivant le raisonnement suivant :
(i) Initialement, tous les nœuds type sont associés au label OK (cela signifie que le langage $\mathcal{L}(\tau) \subseteq \mathcal{L}(\tau')$).
(ii) Chaque primitive utilisée dans l'évolution du schéma original vers le nouveau schéma est considérée.
(iii) Selon la primitive analysée, le label du type peut changer en KO (si $\mathcal{L}(\tau) \cap \mathcal{L}(\tau') = \emptyset$) ou $MAYBE$ (si $\mathcal{L}(\tau) \not\subseteq \mathcal{L}(\tau') \wedge \mathcal{L}(\tau) \cap \mathcal{L}(\tau') \neq \emptyset$). Remarquer que l'analyse d'une suite de primitives peut transformer un nœud $MAYBE$ en nœud OK.
(iv) La phase de propagation, exécuté à la fin de la procédure d'évolution, consiste à propager l'étiquetage aux ancêtres.
(v) Les ancêtres servent de base pour déterminer où la revalidation est nécessaire.

La validation par rapport à un schéma qui a évolué depuis la dernière validation est abordée dans [RS04]. Le scénario considéré comporte un schéma source Υ et un schéma cible Υ' ainsi que des documents va-

[6] Il est raisonnable, même dans le cadre de XML, de considérer que les modifications de schéma restent moins fréquentes que les modifications sur les documents.
[7] Dans [GMR05], un schéma cohérent est définit selon une grammaire $STTG$.

lides[8] par rapport à Υ qui doivent être vérifiés par rapport à Υ'. La question ici est comment le fait de savoir qu'un document est valide par rapport à Υ, peut aider à déterminer s'il est valide par rapport à Υ'. Le but de l'approche étant de profiter des similarités (et des différences) entre schémas pour éviter la revalidation de certaines parties des documents. La méthode est fondée sur la comparaison des sous types et des deux notions suivantes :
Subsumed type : Un type β est *subsumed* par un type β' si $valid(\beta) \subseteq valid(\beta')$.
Disjoint type : Deux types β et β' sont disjoints si $valid(\beta) \cap valid(\beta') = \emptyset$.
Remarquer que $valid(\beta)$ est l'ensemble des arbres valides par rapport à un type β et les types β et β' peuvent être des sous-types des schémas différents. L'algorithme fonctionne sur deux relations qui stockent les sous-types *subsumed* et *disjoints* par rapport aux schémas Υ et Υ'. Ainsi, si pendant la validation un sous-arbre d'un document valid par rapport à β de Υ est testé par rapport à β' de Υ' et si β est *subsumed* par β', alors le sous-arbre n'a pas besoin d'être vérifié, il peut être considéré comme valide. Contrairement, si β et β' sont disjoints ; le document est invalide par rapport à Υ'.

5.8 Conclusions

L'idée de base de notre approche d'évolution de schéma, présentée dans ce chapitre, est :

> Si une mise à jour sur un document XML est incompatible avec le schéma (type) du document, nous pouvons adapter le schéma au document mis à jour en préservant la validité des autres documents (non mis à jour).

Notre approche pour l'évolution de schéma est très originale. La majorité des travaux dans ce domaine considère des opérations capables de transformer le schéma directement. Suite à une de ces transformations sur un schéma Υ, pour obtenir le schéma Υ', les documents valides par rapport à Υ n'ont plus l'assurance de validité par rapport à Υ'. Ces documents doivent alors être revalidés et adaptés par rapport au nouveau schéma. Contrairement à ce raisonnement, notre approche est fondée sur les deux aspects suivants :

1. L'évolution de schéma préserve la cohérence des documents qui étaient valides par rapport au schéma original. Cette solution est intéressante notamment quand les documents sont stockés dans différents sites. Chaque candidat proposé par notre méthode correspond à un schéma (expression régulière, ou programme datalog) plus général que l'original. Néanmoins, nous ne sommes pas intéressés par des solutions triviales. Notre intérêt est de proposer des schémas candidats similaires au schéma initial, en prévoyant une évolution sémantique adaptée à l'application (c'est un administrateur qui prend cette décision sémantique à partir des options données par notre algorithm).
2. L'évolution de schéma est déclenchée par une mise à jour prioritaire sur le document. Cette démarche s'insère dans un contexte d'aide à la maintenance des applications par des administrateurs non spécialistes en informatique mais capables de prendre des décisions sur l'évolution de leurs applications.

Contrairement aux travaux cités dans la section 5.7, l'évolution de schémas proposée par `GREC` et `dGREC` ne considère pas un schéma XML en entier, c'est-à-dire, elle prend en compte l'évolution de chaque sous schéma représenté par une expression régulière et non l'arborescence qui doit définir la structure du document XML. En d'autres termes les mise à jour consiste à insérer ou ajouter une position dans un mot. Cela peut être considéré comme une limitation de notre méthode qui proposera ainsi des solutions locales à un utilisateur, sans lui donner une vision d'ensemble du schéma.

Par contre, un programme datalog représente tout le schéma. Même si l'approche $\texttt{dGREC}^{Dat}$ suit les lignes des `dGREC`, il permet de considérer l'insertion ou la suppression d'un sous arbre dû aux aspects suivants : (*i*) Des règles datalog sont ajoutées à un programme $\mathcal{P}$ pour exprimer l'évolution d'une expression régulière définissant le schéma associé à une position p (ces règles expriment la transformation de E en E' dans `dGREC`). (*ii*) Le programme datalog correspondant au nouveau schéma associé à la position d'insertion $p.i$ est ajouté à $\mathcal{P}$.

[8] De plus, une version plus générale où le document, lui aussi, a été mis à jour est également abordée dans [RS04].

Chapitre 6

Mises à jour multiples et les contraintes d'intégrité : validation incrémentale

Comme dans les bases de données traditionnelles, les données XML peuvent être spécifiées par des contraintes de types et des contraintes d'intégrité. D'une façon générale, le type impose des contraintes sur la structure du document, alors que les contraintes d'intégrité imposent des restrictions sur les valeurs. Puisque XML est devenu le standard pour les données du Web et un modèle pour l'intégration de données, les contraintes d'intégrité sont essentielles non seulement pour la conception du schéma, l'optimisation des requêtes et l'efficacité de stockage et d'accès (comme dans le modèle relationnel) mais aussi dans la préservation de la sémantique des données lors d'un changement de modèle ou dans la détection des incohérences lors d'une intégration de données [Fan05].

Quelques langages pour la conception des schémas XML permettent la spécification des certaines contraintes d'intégrité. La proposition de XML Schema (XSD) permet de définir des clés et des clés étrangères. La proposition des attributs ID et IDREF d'une DTD comme, respectivement, clés et clés étrangères d'un document XML est une des premières tentatives d'introduction des contraintes d'intégrité dans le cadre de XML. Les limites de ces attributs sont bien connues : (1) Un attribut ID doit avoir une valeur unique, mais parmi tous les attributs ID du document XML. (2) L'utilisation des attributs ID comme clé, nous limite au cas des clés unaires. (3) Un seul attribut ID par élément peut être spécifié alors que, dans la pratique plusieurs clés peuvent être souhaitées. (4) La portée d'un attribut IDREF(S) n'est pas claire puisqu'une référence IDREF peut correspondre à n'importe quel ID dans le document.

L'extension des contraintes d'intégrité du relationnel vers XML n'est pas triviale. En particulier, le langage pour l'expression des contraintes d'intégrité XML doit être plus riche que celui utilisé par les contraintes relationnelles. La nature hiérarchique des documents XML demande l'introduction de deux catégories de contraintes. Les contraintes *absolues* vérifiées par tout le document XML et les contraintes *relatives* vérifiées par une partie d'un document. Ainsi, dans la première partie du chapitre nous considérons le problème de l'expression des contraintes d'intégrité. Pour spécifier les parties d'un document XML concernées par une contrainte, nous avons besoin d'un langage de chemin. Comme XPath est un langage permettant la sélection des nœuds d'un arbre XML, il est naturel de le considérer pour cette tâche. Néanmoins non seulement l'expression des contraintes d'intégrité ne nécessite pas tout le pouvoir d'expression de XPath mais aussi son utilisation s'avère très compliquée [BFK03, BDF$^+$03]. Les langages de chemin proposés correspondent, en général, à des fragments de XPath. Ce chapitre fait une synthèse de nos travaux, tout en proposant un nouveau regard sur le langage de définition des contraintes. Ainsi, dans ce mémoire, nous introduisons les requêtes arbre (inspirées de [BFK03, Deb05, GI06, GKS04, HT06]) pour définir les contraintes en proposant un parallèle entre ce formalisme et les chemins linéaires normalement utilisés (y compris dans nos articles [ABH$^+$04, BCH$^+$07, BHL07]).

Un autre point important dans le traitement des contraintes d'intégrité XML est la question de l'égalité. La valeur d'un nœud est spécifiée non seulement par son label mais aussi par les valeurs de ses descendants. De plus, les contraintes ont souvent besoin de faire la différence entre l'égalité de nœuds

(qui met en jeu la position du nœud) et l'égalité de valeurs (qui prend en compte l'étiquette et le type du nœud ainsi que ses descendants). Dans ce chapitre nous définissons ce deux types d'égalité.

Après avoir traité ces deux aspects importants (le langage pour la spécification de contraintes et la question de l'égalité), nous présentons rapidement les dépendances fonctionnelles (XFD), les dépendances d'inclusion (XID), les clés et les clés étrangères. En utilisant un formalisme homogène nous pouvons aborder de façon simple les différentes propositions de la littérature.

La fin du chapitre présente notre méthode de validation incrémentale par rapport aux contraintes d'intégrité. Dans ce mémoire, nous présentons l'algorithme de base en expliquant rapidement la méthode utilisée. L'algorithme est fondé sur celui présenté dans le chapitre 4 pour la validation des contraintes de schéma. Ainsi, la validation de contraintes d'intégrité est faite dans un parcours du document XML. En effet, nos travaux de recherche dans ce domaine ont commencé par la proposition d'une méthode de validation des clés et des clés étrangères, *from scratch* dans [BHM03] et ensuite incrémentale dans [ABH$^+$04]. Dans ces travaux la validation du schéma était faite en parallèle à la validation des clés par un automate d'arbre auquel nous avons intégré des fonctions de sorties pour remonter les valeurs de façon *bottom-up*. L'utilisation d'une grammaire d'attribut à la place des fonctions associées à l'automate a donné plus de lisibilité à notre méthode [BCH$^+$07].

Plusieurs points abordés dans ce chapitre font partie de la thèse de Maria Adriana Lima (ex Abrão) que je co-encadre. Dans sa thèse, en plus des détails concernant la méthode de validation incrémentale par rapport aux contraintes d'intégrité, Maria Adriana Lima doit réunir dans un formalisme homogène les propositions actuelles concernant les principales contraintes d'intégrité pour XML. Dans ce mémoire, je donne un aperçu de cette démarche (qui est déjà présentée dans [BHL07]) tout en proposant un autre formalisme - les requêtes arbre. Mon but est de faire un parallèle entre ces deux formalismes ainsi que d'ouvrir des perspectives de recherche, en particulier vers l'adaptation de l'approche [GI06] aux contraintes d'intégrité.

6.1 L'expression des contraintes d'intégrité : langages de requêtes et types d'égalité

Pour exprimer les contraintes d'intégrité nous avons besoin de désigner certaines parties d'un document XML pour ensuite pouvoir les comparer en testant, par exemple, si ces parties sont identiques. Pour cela nous devons d'abord définir un langage de requête et ensuite définir une ou plusieurs notions d'égalité.

La désignation d'une partie d'un document est, en général, faite via un langage de chemin, allant de la séquence d'étiquettes à des langages plus expressifs, comme XPath [XPab]. Les travaux sur les contraintes d'intégrité utilisent des langages de chemins plutôt simples. Dans nos travaux [BHL07], nous avons considéré un langage de chemin similaire au langage de [BDF$^+$03].

Langage de chemin PL : Dans ce mémoire un chemin en PL est défini par $q ::= \epsilon \mid l \mid q/q \mid q//l$ où ϵ représente le chemin vide[1], l est une étiquette dans Σ, le symbole "/" est l'opération de concaténation et "//" est un *wildcard* qui représente n'importe quelle séquence finie (éventuellement vide) d'étiquettes de nœuds.

Néanmoins, même si, en général, les langages de chemins utilisés dans la plupart des travaux définissent des chemins linéaires, la manipulation des contraintes d'intégrité demande l'examen de différentes branches d'un arbre XML. Autrement dit, dans le contexte des contraintes d'intégrité nous sommes souvent intéressés par un "chemin arborescent". Dans la suite nous introduisons les requêtes arbre pour ensuite définir des chemins linéaires majoritairement utilisés pour les contraintes. Nous comptons ainsi faire un parallèle entre ces deux manières d'exprimer les contraintes.

[1] À ne pas confondre avec la position de la racine d'un arbre. Nous rappelons que $l/\epsilon = \epsilon/l = l$ et $\epsilon//l = //l$.

6.1.1 Les requêtes arbre

La notion de requête arbre que nous introduisons dans ce mémoire est proche du *tree pattern* utilisé dans [BFK03] et est une version simplifiée des propositions trouvées dans [Deb05, GI06, GKS04, HT06]. Notre requête arbre est définie comme un arbre où chaque arc est associé à une étiquette. Nous supposons seulement deux étiquettes d'arc et nous considérons que les nœuds à sélectionner sont les nœuds feuilles de la requête arbre. Selon la contrainte à spécifier nous verrons que d'autres nœuds spéciaux peuvent être nécessaires.

Définition 6.1.1 - Requête arbre : Soit Σ un alphabet. Une requête arbre `Query` est un n-uplet $\texttt{Query} = (D_q, q, Lab_q, S_q)$ où (D_q, q) est un arbre sur Σ, Lab_q est une fonction qui associe à chaque couple (u, v) tel que u est le père de v une étiquette dans l'ensemble $\{/, //\}$ et S_q est le n-uplet des nœuds sélectionnés, c'est-à-dire, S_q est de la forme $[\mu_1, \ldots, \mu_n]$ où chaque μ_i correspond à une des n feuilles de (D_q, q). □

Étant donnée une requête arbre, nous nous intéressons à un parcours (instantiation) arborescent, introduit dans la définition ci-dessous.

Définition 6.1.2 - Parcours arborescent : Soit un arbre $T = (D, t)$. Un *parcours arborescent* sur T est un couple $\Im = (D^{\Im}, t^{\Im})$ tel que $D^{\Im} \subseteq D$ et $D^{\Im}$ est fermé par préfixe et pour toute position $p \in D^{\Im}$, $t^{\Im}(p) = t(p)$. □

L'arbre T est lui même le plus grand parcours arborescent sur T, mais, en général, un parcours arborescent n'est pas un arbre car $D^{\Im}$ peut ne pas respecter toutes les conditions nécessaires pour être un domaine d'arbre (définition 2.1.1).

Exemple 6.1.1 Soit l'arbre XML de la figure 6.1. Le parcours $\Im_0 = (D_0^{\Im}, t_0^{\Im})$ où :
$D_0^{\Im} = \{\epsilon, 0, 0.0, 0.0.2, 0.0.2.0, 0.0.2.1, 0.0.2.0.0, 0.0.2.1.0\}$ et
$t_0^{\Im} = \{(\epsilon, fac), (0, students)(0.0, student), (0.0.2, courseTaken), (0.0.2.0, courseCode), (0.0.2.1, deptName)$, (0.0.2.0.0, data), (0.0.2.1.0, data)$\}$ est un parcours arborescent sur T. Remarquer que $(D_0^{\Im}, t_0^{\Im})$ n'est pas un arbre car les positions 0.0.0 et 0.0.1 n'existent pas dans $D^{\Im}$. □

L'instantiation (ou le plongement) d'une requête arbre, que nous appelons la projection d'un arbre T sur une requête `Query`, notée $\Pi_{\texttt{Query}}(T)$, correspond à un parcours arborescent. Cette notion est à rapprocher de celle utilisée dans [BCCN06]. Définir $\Pi_{\texttt{Query}}(T)$ signifie définir un morphisme entre `Query` et T (comme dans [Deb05, GI06]).

Définition 6.1.3 - Projection d'un arbre T sur une requête `Query` : Soient un arbre $T = (D, t)$ et une requête arbre $\texttt{Query} = (D_q, q, Lab_q, S_q)$. Une projection $\Pi_{\texttt{Query}}(T)$ (ou un plongement) est une fonction injective $h : D_q \to D$ vérifiant les propriétés suivantes :

- La racine de la requête `Query` est associée à la racine de l'arbre T, c'est-à-dire, $q(\epsilon) = t(\epsilon)$.
- Chaque position $p \in D_q$ est associée à une *unique* position $h(p) \in D$ c'est-à-dire, $q(p) = t(h(p))$.
- Pour chaque couple (v, u) tel que $v, u \in D_q$, $v \prec_{ch} u$ et $Lab_q(v, u) = /$ il existe un couple $(h(v), h(u))$ tel que $h(v), h(u) \in D$ et $h(v) \prec_{ch} h(u)$.
- Pour chaque couple (v, u) tel que $v, u \in D_q$, $v \prec_{ch} u$ et $Lab_q(v, u) = //$ il existe un couple $(h(v), h(u))$ tel que $h(v), h(u) \in D$ et $h(v) \prec^*_{ch} h(u)$.
- Pour toutes les positions $p, p' \in D_q$, si $p \prec^*_{ch} p'$ alors $h(p) \prec^*_{ch} h(p')$; si $h(p) = h(p')$ alors $p = p'$. □

Les nœuds sélectionnés composent ainsi un n-uplet contenant $h(\mu_1), \ldots, h(\mu_n)$, c'est-à-dire, les positions sélectionnées par la requête. Dans plusieurs cas, nous nous intéressons aux valeurs associées à ces positions.

Exemple 6.1.2 Soit l'arbre XML de la figure 6.1 et la requête arbre de la figure 6.2. Le parcours arborescent $\Im_0 = (D_0^{\Im}, t_0^{\Im})$ de l'exemple 6.1.1 est celui obtenu par une projection de T sur `Query`. Ce parcours est montré en trait continu (bleu) dans la figure 6.2. □

Les requêtes arbre permettent la définition des contraintes d'intégrité. Néanmoins dans la grande majorité des travaux (y compris le nôtre [ABH+04, BCH+07]) les contraintes sont définies à partir de chemins linéaires (non arborescents). Pour faire un parallèle entre ces deux façons de voir les choses, dans la suite nous rappelons (en adaptant) des définitions concernant les chemins linéaires et nous proposons une nouvelle définition d'une projection $\Pi_{\texttt{Query}}(T)$, similaire à la définition 6.1.3, mais constructive et permettant des liens avec les chemins linéaires.

En effet, notre requête arbre est une sorte d'abréviation représentant une collection de chemins exprimés dans un langage de chemin (comme PL). Autrement dit, même si les requêtes arbre sont des requêtes n-aires dont l'évaluation avec des méthodes plus sophistiquées pourrait être envisagées, dans ce mémoire, cela n'est pas pris en compte. En fait, ici, les requêtes arbre indiquent un ensemble de requêtes unaires (chemins) dont l'évaluation peut être faite requête par requête.

Dans la suite nous introduisons aussi les notions nécessaires pour pouvoir traiter les informations incomplètes. En effet, puisque les documents XML ont des parties optionnelles, l'importance du traitement de l'information incomplète n'est pas moindre. Un document XML peut être valide par rapport à un schéma tout en ayant des branches (optionnelles) qui manquent. Dans l'analyse de contraintes d'intégrité, les informations manquantes peuvent correspondre à des données concernant une contrainte et il faut ainsi prendre cela en considération.

6.1.2 Les chemins linéaires

Commençons donc par rappeler la définition d'un chemin linéaire simple.

Définition 6.1.4 - Chemin linéaire simple : Un chemin (linéaire) simple P est une séquence de labels $l_1/\ldots/l_n$, où $n \geq 1$. Le chemin simple P est dit un préfixe du chemin $Q = q_1/\ldots/q_m$, noté $P \subseteq Q$, si $n \leq m$ et $l_1 = q_1, \ldots, l_n = q_n$. Deux chemins P et Q sont égaux, noté $P = Q$ si P est un préfixe de Q et Q est un préfixe de P. Le chemin simple P est dit un préfixe strict du chemin Q, noté $P \subset Q$, si P est un préfixe de Q et $P \neq Q$. □

REMARQUE - Dans la suite du chapitre, nous adoptons la nomenclature suivante : *chemin* correspond à une expression (dans un langage donné) pouvant contenir des *wildcards*, alors que *chemin simple* est une expression sans *wildcards*. Remarquer qu'un chemin peut être traduit dans un ensemble, éventuellement infini, de chemins simples.

Exemple 6.1.3 Soit le document XML partiellement illustré dans la figure 6.1 et soit le langage PL. Le chemin $P_0 = fac//name/data$ détermine toutes les données associées à un élément *name* à savoir, les noms des étudiants et des cours. Le chemin $P_1 = fac/students/student$ est un préfixe strict du chemin $P_2 = fac/students/student/courseTaken$. □

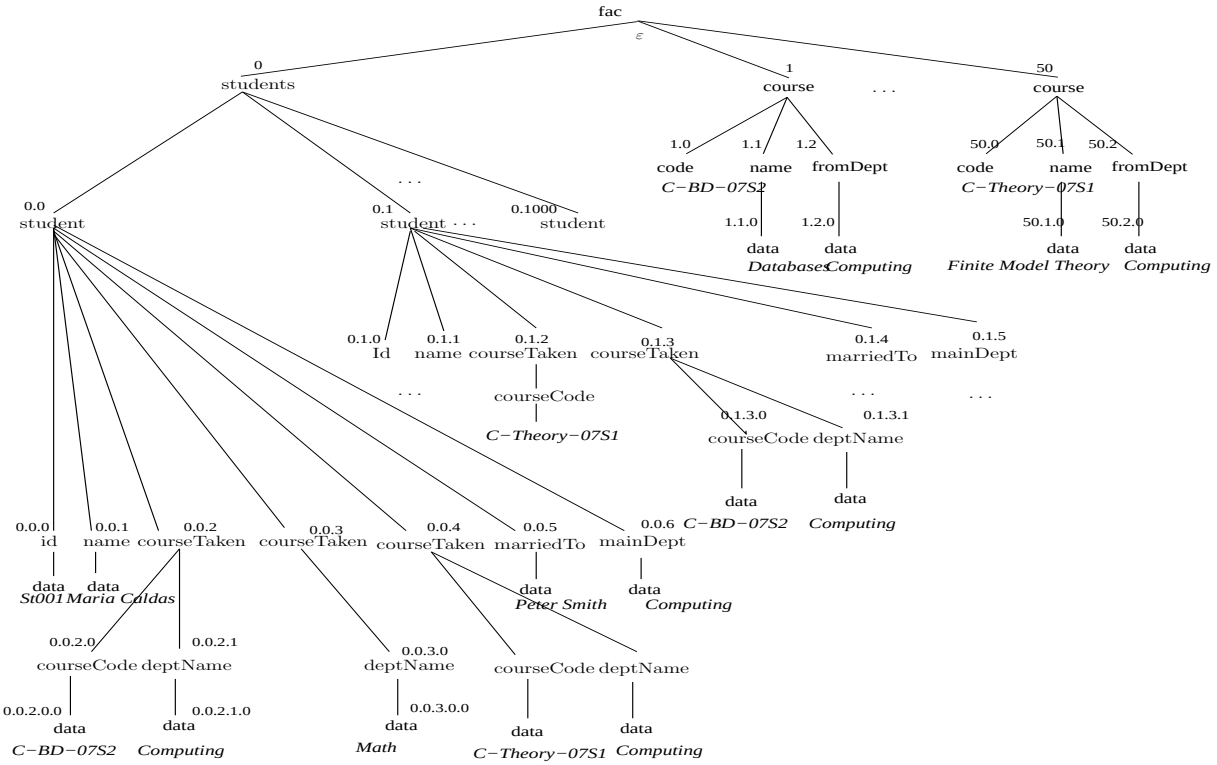


FIG. 6.1: Arbre XML.

Étant donné un chemin P et un arbre T, trouver une instance de P sur T signifie trouver les parcours dans T qui respectent P. Dans un arbre T, il peut y avoir zéro ou plusieurs instances d'un chemin P donné. Par exemple selon les indications de la figure 6.1, le document XML en question possède 50 instances du chemin $fac/course/code$, qui sont $\varepsilon/1/1.0$, $\varepsilon/2/2.0$, $\ldots\varepsilon/50/50.0$.

Définition 6.1.5 - Parcours linéaire simple : Soit un arbre $T = (D, t)$. Un *parcours (linéaire simple)* sur T est une séquence de positions $v_1/\ldots/v_n$ telle que pour tout v_i $(1 < i \leq n)$, $v_i \in D$ et v_i est *fils* de v_{i-1}. Deux parcours $I_1 = v_1/\ldots/v_n$ et $I_2 = v'_1/\ldots/v'_n$ sont distincts si $v_i \neq v'_i$ pour un i, tel que $1 \leq i \leq n$. Un parcours $I_1 = v_1/\ldots/v_n$ est un *préfixe* d'un autre parcours $I_2 = v'_1/\ldots/v'_m$ si $n \leq m$ et $v_i = v'_i$ pour tout i, $1 \leq i \leq n$. La fonction $LastPosition$ retourne la dernière position d'un parcours $(LastPosition(I_1) = v_n)$. □

Le langage que nous avons considéré dans [ABH+04, BHL07] nous permet de construire des expressions de chemin correspondant à des cas spéciaux d'expressions régulières. Nous pouvons donc considérer qu'un chemin P définit un automate d'état fini A_P. Trouver les instances d'un chemin P sur un arbre XML t correspond à trouver les parcours appartenant au langage $L(A_P)$ défini par A_P.

Définition 6.1.6 - Instance d'un chemin P sur un arbre T : Soit P un chemin dans un langage de chemin donné et soit A_P l'automate d'état fini défini par P. Soit $T = (D, t)$ un arbre XML. Un parcours $I = v_1/\ldots/v_n$ dans t est une instance de P sur T ssi la concaténation $t(v_1).\ldots.t(v_n)$ est un mot du langage $L(A_P)$ (par abus de notation nous disons $I \in L(A_P)$). On note $Instances(P, t)$ le langage contenant toutes les instances d'un chemin P dans un arbre T.

Un parcours $I = v_1/\ldots/v_k$ dans t est une *instance incomplète* de P sur T ssi il est une instance d'un préfixe strict de P qui respecte les conditions suivantes :
(i) $\hat{\delta}(s_0, t(v_1).t(v_2)\ldots t(v_k)) = s$ où, pour l'automate d'état fini A_P, $\hat{\delta}$ est sa fonction de transition étendue[2], s_0 est son état initial et s est un état non final de A_P et
(ii) il n'existe aucun fils $v_k.i$ de v_k tel que $\delta(s, t(v_k.i))$ est une transition de A_P. □

Un document XML peut être incomplet, c'est-à-dire, il peut ne pas contenir des instances pour tous les chemins P d'un ensemble $Path_T$ donné, alors qu'il contient des instances pour un préfixe de P. Même si le document est conforme à un schéma, ce schéma peut comporter des parties optionnelles. Le traitement correct des informations incomplètes est fondamental, aussi bien pour le calcul des requêtes que pour la vérification des contraintes d'intégrité.

Exemple 6.1.4 Soit T l'arbre XML de la figure 6.1. Soit le chemin simple $fac/students/student/course\ Taken/CourseCode/data$. Le parcours $\epsilon/0/0.0/0.0.2/0.0.2.0/0.0.2.0.0$ est une instance de P sur T. Le parcours $\epsilon/0/0.0/0.0.3$ est une instance incomplète de P sur T. □

Une requête arbre peut être traduite par un ensemble de chemins ayant un préfixe commun et en particulier par un ensemble fini de chemins simples fermé par préfixe qui correspond à la définition de *motif* utilisé dans [BHL07, VLL04]. En effet, dans [BHL07] nous considérons l'existence d'un *motif maximal*, dont la définition est similaire à celle de l'ensemble de chemins légaux de [VLL04]. Le motif maximal est un ensemble fini de chemins (fermé par préfixe et donc tous les chemins ont un préfixe commun) qui décrit tous les chemins possibles dans un arbre XML. Les motifs applicables sur un arbre XML sont des sous ensembles d'un motif maximal donné. Nous nous intéressons par de motifs (applicables) définis à partir d'une requête (contrainte).

En général, étant donnée une requête arbre $\texttt{Query} = (D_q, q, Lab_q, S_q)$ nous serons intéressé par l'ensemble $LongestPath_{\texttt{Query}}$, l'ensemble de chemins de `Query` qui ne sont préfixes[3] d'aucun autre chemin de `Query`. Plus précisément, $LongestPath_{\texttt{Query}}$ contient les chemins linéaires composés par la concaténation des labels des nœuds et des arcs obtenus en partant de la racine de la requête `Query` jusqu'à chacune de ses feuilles, comme illustré par l'exemple ci-après.

Exemple 6.1.5 Soit la requête arbre `Query` de la figure 6.2. L'ensemble $LongestPath_{\texttt{Query}}$ contient deux chemins, à savoir $fac//student/courseTaken/deptName/data$ et
$fac//student/courseTaken/courseCode/data$.

[2]La fonction de transition étendue ([HMU01]) décrit le résultat (l'état) obtenu via un parcours sur l'automate d'état fini qui commence par un état donné, en lisant une suite de symboles d'entrée. La fonction $\hat{\delta}$ est construite sur la fonction de transition δ : $\hat{\delta}(s, \epsilon) = s$ (si s est un état de l'automate et aucun symbole est lu) et $\hat{\delta}(s, w) = \delta(\hat{\delta}(s, x), a)$ (où $w = xa$).

[3]Pour simplifier, un préfixe est défini de façon complètement syntaxique, c'est-à-dire, une sous chaîne d'une chaîne donnée. Ainsi, les chemins $r/A/B$ et $r//B$ ont r comme unique préfixe commun.

L'ensemble $\{fac, fac/students, fac/students/student, fac/students/student/courseTaken,$
$fac/students/student/courseTaken/courseCode, fac/students/student/courseTaken/deptName,$
$fac/students/student/courseTaken/courseCode/data,$
$fac/students/student/courseTaken/deptName/data\}$

correspond au motif (selon [BHL07]) défini à partir de $LongestPath_{\texttt{Query}}$ et ainsi associé à `Query`. Remarquer que l'ensemble de labels composant ces chemins simples est le projecteur de [BCCN06]. □

Un parcours arborescent est composé d'un ensemble non vide de parcours simples ayant un préfixe commun. Trouver une projection $\Pi_{\texttt{Query}}(T)$ d'un arbre T sur une requête `Query` correspond à trouver un parcours arborescent qui est, en fait, composé des parcours (simples) ayant un préfixe commun. La définition suivante représente une nouvelle vision (constructive) de la définition 6.1.3 en s'appuyant sur la notion de chemins linéaires. En effet, cette définition est équivalente à la notion d'instance de motif définie dans [BHL07] et similaire à la notion de projection proposée dans [BCCN06][4].

Définition 6.1.7 - Projection d'un arbre T sur une requête `Query` (Version constructive) : Soit un arbre $T = (D, t)$. Soit une requête arbre $\texttt{Query} = (D_q, q, Lab_q, S_q)$ tel que $q(\epsilon) = t(\epsilon)$. Soient $LongestPath_{\texttt{Query}}$ l'ensemble de chemins de `Query` qui ne sont préfixes d'aucun autre chemin de `Query`. Soit *SetPathInst* un ensemble de parcours qui vérifie les conditions suivantes :

1. Pour tout chemin $P \in LongestPath_{\texttt{Query}}$ il existe un et un seul parcours $parc \in Instances(P, t)$ dans *SetPathInst*.
2. Pour tout parcours $parc \in SetPathInst$ il existe un chemin $P \in LongestPath_{\texttt{Query}}$ tel que $parc \in Instances(P, t)$.
3. Pour tous parcours $parc$ et $parc'$ dans *SetPathInst*, tel que $parc \in Instances(P, t)$ et $parc' \in Instances(P', t)$, le plus long préfixe commun de $parc$ et $parc'$ correspond à un même parcours de P_0 dans T (P_0 étant le plus grand (chemin) préfixe commun à P et P').

Une projection de T sur `Query`, notée $\Pi_{\texttt{Query}}(T)$, définit un parcours arborescent $(D^{\Im}, t^{\Im})$ sur T où :

- $D^{\Im} = \bigcup_{parc \in SetPathInst}\{p \mid p \text{ est une position dans } parc\}$
- $t^{\Im}(p) = t(p)$ □

Autrement dit, $\Pi_{\texttt{Query}}(T)$ définit le parcours arborescent qui contient les parcours simples de *SetPathInst*. Remarquer que dans le cas des informations incomplètes, la définition 6.1.7 est étendue pour que l'ensemble *SetPathInst* contienne aussi des parcours incomplets (définition 6.1.6).

Exemple 6.1.6 Comme dans l'exemple 6.1.5, soit l'arbre XML de la figure 6.1 et la requête arbre de la figure 6.2. Non seulement le parcours arborescent $\Im = (D_0^{\Im}, t_0^{\Im})$ (en bleu dans la figure 6.2) de l'exemple 6.1.1 correspond à une projection de T sur `Query`, mais aussi le parcours *incomplet* $\Im_1 = (D_1^{\Im}, t_1^{\Im})$ (trait avec tirets et points, en vert, dans la figure 6.2) où $D_1^{\Im} = \{\epsilon, 0, 0.0, 0.0.3, 0.0.3.0, 0.0.3.0.0\}$ et $t_1^{\Im} = \{(\epsilon, fac), (0, students)(0.0, student), (0.0.3, courseTaken),$
$(0.0.3.0, deptName),(0.0.3.0.0, \text{data})\}$.

Remarquer que $\Im_2 = (D_2^{\Im}, t_2^{\Im})$ (trait avec tirets, en rouge, dans la figure 6.2) où $D_2^{\Im} = \{\epsilon, 0, 0.0, 0.0.2,$ $0.0.2.0, 0.0.2.0.0, 0.0.4, 0.0.4.0, 0.0.4.0.1\}$ et $t_2^{\Im} = \{(\epsilon, fac), (0, students)(0.0, student), (0.0.2, courseTaken),$ $(0.0.2.0, courseCode), (0.0.2.0.0, data), (0.0.4, courseTaken), (0.0.4.0, deptName), (0.0.4.0.1, data)\}$ n'est pas une projection. En effet, le plus grand chemin préfixe commun P_0 de la définition 6.1.7 est le chemin $fac//student/courseTaken$ et donc $\mathcal{I}_2$ ne respecte pas la définition 6.1.7. □

Soient `Query` une requête arbre et T un arbre. En suivant la définition 6.1.7, le n-uplet S_q de `Query` contient les positions atteintes par les chemins linéaires dans $LongestPath_{\texttt{Query}}$. Nous considérons maintenant une relation où chaque n-uplet correspond aux valeurs déterminées par une projection $\Pi_{\texttt{Query}}(T)$, c'est-à-dire, aux valeurs associées aux positions composant le n-uplet S_q. La définition ci-dessous formalise cette notion qui nous permet de comparer les différentes valeurs obtenues par une requête sur un arbre. Remarquer que nous considérons l'extension des définitions 6.1.3 et 6.1.7 où les informations incomplètes sont prises en compte.

Définition 6.1.8 - N-uplet résultat de `Query` sur T : Soient `Query` une requête arbre et son ensemble $LongestPath_{\texttt{Query}}$. Soit $\Pi_{\texttt{Query}}(T)$ une projection d'un arbre T sur `Query`. Selon la définition 6.1.7, $\Pi_{\texttt{Query}}(T)$

[4]Dans [BCCN06] le résultat de la projection est un arbre obtenu par l'ajout du l'arbre vide sur les branches ne correspondant pas à un chemin du projecteur. Notre définition pourrait être adaptée de façon à devenir équivalente à celle de [BCCN06].

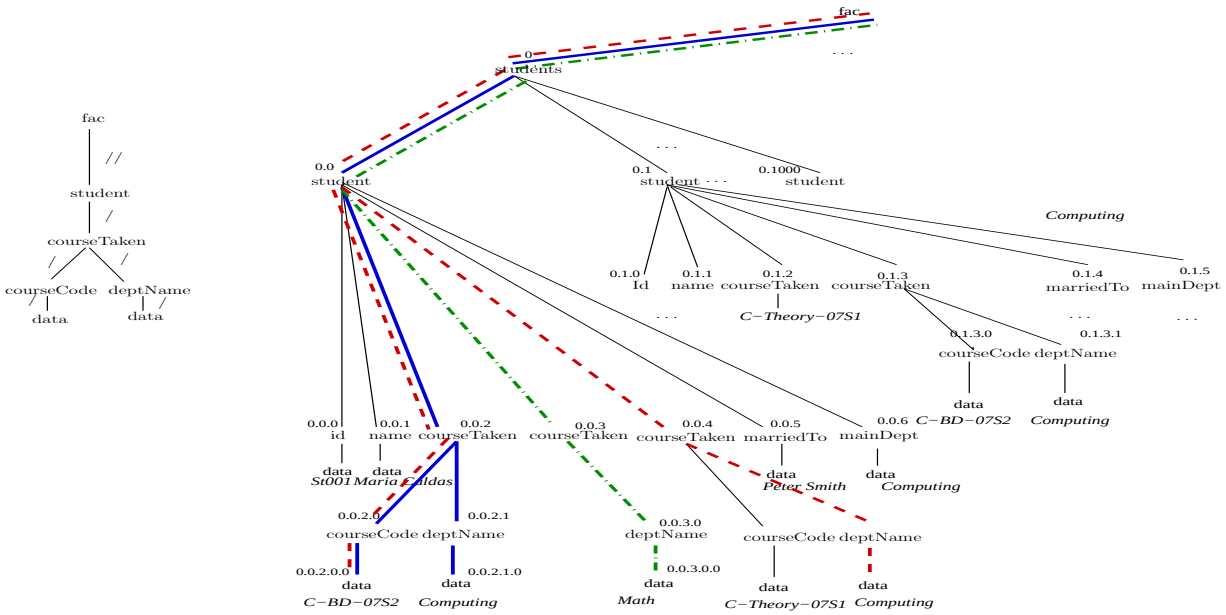


FIG. 6.2: Requête arbre et sa projection sur l'arbre. Le parcours en rouge n'est pas une projection.

contient une et une seule instance de chaque $P_j \in LongestPath_{\texttt{Query}}$ (nous notons I_j l'instance du chemin P_j, $1 \leq j \leq n$). Le n-uplet résultat de `Query` sur T par rapport à $\Pi_{\texttt{Query}}(T)$, noté $\tau_{\texttt{Query}}[P_1, \ldots, P_n]$, est tel que :

$$\tau_{\texttt{Query}}[P_1, \ldots, P_n] = [value(LastPosition(I_1)), \ldots, value(LastPosition(I_n))]$$

où la fonction *value* fournit pour le dernier nœud p déterminé par I_j : (1) sa valeur, si elle existe ou (2) sa position, si p n'est pas un nœud associé à une valeur[5] ou (3) une valeur *null* si I_j est une instance incomplète. □

Le n-uplet $\tau_{\texttt{Query}}[P_1, \ldots, P_n]$ est construit selon la perspective nommée des bases de données relationnelles [AHV95] (c'est-à-dire que les noms des attributs composant le n-uplet sont connus). Bien entendu, deux n-uplets $\tau_1[P_1, \ldots, P_n]$ et $\tau_2[P_1, \ldots, P_n]$ sont égaux ssi $\forall i \in [1 \ldots n], \tau_1[P_i] = \tau_2[P_i]$.

Exemple 6.1.7 Soit l'arbre XML de la figure 6.1 et la requête arbre de la figure 6.2. Considérons maintenant les projections $\Pi_{\texttt{Query}}(T)$ converties en n-uplet, c'est-à-dire, les n-uplets résultat de `Query` sur T. La relation ci-dessous représente ces n-uplets (sans les doublons). Remarquer que dans la figure 6.1 nous n'avons pas représenté tout le document XML . Nous supposons que la relation ci-dessous représente certains n-uplets obtenus par les projections $\Pi_{\texttt{Query}}(T)$.

fac//student/courseTaken/courseCode/data	*fac//student/courseTaken/deptName/data*
C-DB-07S2	Computing
null	Math
C-Theory-07S1	Computing
C-Theory-07S1	*null*
C-Prog-07S2	Math
C-Logic-07S2	*null*

□

Définition 6.1.9 - Projections de `Query` qui coïncident sur un chemin P : Soient $\Pi^1_{\texttt{Query}}(T)$ et $\Pi^2_{\texttt{Query}}(T)$ deux projections d'un arbre T sur une requête `Query`. Nous disons que $\Pi^1_{\texttt{Query}}(T)$ et $\Pi^2_{\texttt{Query}}(T)$ *coïncident sur un chemin simple* P (P partant de la racine) ssi I est le (seul) parcours de P dans $\Pi^1_{\texttt{Query}}(T)$, J est le (seul) parcours de P dans $\Pi^2_{\texttt{Query}}(T)$ et $I = J$. □

Exemple 6.1.8 Soit l'arbre XML de la figure 6.1 et la requête arbre de la figure 6.2. Soient les projections considérées dans les exemples 6.1.1 et 6.1.6 définissant les parcours arborescents $\Im$ (en gras dans la figure 6.2) et $\Im_1$ (trait avec des tirets et point dans la figure 6.2). Ces projections coïncident sur

[5]Dans la représentation d'un arbre XML que nous avons prise en compte les seuls nœud associés à une valeur sont les nœuds *data* (section 2.2, figure 2.2).

$fac/students/student/$, et évidement, sur tous les préfixes de ce chemin. □

6.1.3 Les types d'égalité

Après avoir récupéré certaines parties d'un document, nous devons les comparer. Nous pouvons prévoir des contraintes qui comparent les positions d'un arbre, ou les valeurs associées aux positions *data*, ou des valeurs *null*... Selon les caractéristiques des informations à comparer, nous devons utiliser différents types d'égalité. Cela donne différentes sémantiques aux travaux sur les contraintes d'intégrité.

Dans nos travaux [BHL07, Lim07] nous proposons deux types d'égalité. En plus de l'égalité pour indiquer que deux positions d'un arbre sont les mêmes (égalité de nœuds) utilisée, par exemple, dans la définition 6.1.5, nous introduisons l'égalité en valeur qui prend en compte si une position représente un nœud du type élément ou data (ou attribut).

Définition 6.1.10 - Égalité en valeur : Deux nœuds de positions p et q sont égaux en valeur, égalité notée $p =_V q$, si les conditions suivantes sont respectées :
(i) $t(p) = t(q)$ (ils ont la même étiquette) ;
(ii) si p et q sont des nœuds *data* (c'est-à-dire, nœuds associés à des valeurs) alors ils ont la même valeur sur leur donnée[6] ;
(iii) si p et q sont des nœuds *élément* alors il existe une fonction bijective entre les fils de p et les fils de q qui associe chaque position $p.i$ à une position $q.j$ telle que $p.i =_V q.j$ □

Dans la figure 6.1 le nœud *CourseTaken* de la position 0.0.2 est égal en valeur au nœud *CourseTaken* de la position 0.1.3. Remarquer que l'égalité par valeur ne prend pas en compte l'ordre des fils d'un nœud. La définition suivante permet d'indiquer que deux nœuds concordent sans préciser par quelle égalité.

Définition 6.1.11 - Concordance de nœuds : Soient deux nœuds n_1 et n_2 de même étiquette dans un arbre T. Les nœuds n_1 et n_2 concordent, ce qui se note $n_1 =_E n_2$, pour $E \in \{V, N\}$, s'il sont égaux en valeur ($n_1 =_V n_2$) ou s'ils représentent le même nœud ($n_1 =_N n_2$). Nous notons $(n_1, n_2) =_{E_1,E_2} (n'_1, n'_2)$ pour exprimer que $n_1 =_{E_1} n'_1$ et $n_2 =_{E_2} n'_2$. □

Par défaut nous considérons l'égalité de valeurs. De même, les chemins finissant dans un nœud *data* (ou *attribut*) peuvent être associés seulement à l'égalité de valeurs.

6.2 Les contraintes d'intégrité en XML

Nous considérons maintenant différentes contraintes d'intégrité sur un document XML. Dans [BHL07, Lim07], un panorama, guidé par une syntaxe homogène est proposé. Dans ce mémoire nous présentons un résumé du panorama en question et nous montrons aussi comment les requêtes arbre peuvent représenter chaque contrainte. Le tableau ci-dessous donne un aperçu des principales contraintes traitées dans [BHL07, Lim07] en référençant la requête arbre correspondante. Nous considérons les dépendances fonctionnelles (XFD), les dépendances d'inclusion (XID), les clés (XKey) et les clés étrangères (XForeignKey). D'autres contraintes (par exemple les contraintes d'inversion et les contraintes de domaine) sont traitées dans [BHL07, Lim07].

Table 6.2.1 - ***Table de contraintes d'intégrité :***

Contraintes	***Syntaxe***	***Requête arbre***
XFD	$(C, (\{P_1\ [E_1], \ldots, P_k\ [E_k]\} \rightarrow Q\ [E]))$	*figure 6.3(a)*
XID	$(C, (\{P_1, \ldots, P_k\} \subseteq \{Q_1, \ldots, Q_k\}))$	*figure 6.3(b)*
XKey	$(C, (Tg, \{P_1, \ldots, P_k\}))$	*figure 6.3(c)*
XForeignKey	$(C, (Tg^R, \{P_1^R, \ldots, P_k^R\}) \subseteq (Tg, \{P_1, \ldots, P_k\}))$	*figure 6.3(d)*

La table 6.2.1 montre que les chemins définissant des contraintes sont partagés en sous-chemins (C, P, Q, Tg) qui sont, eux aussi, des chemins décrits dans le langage PL (défini au début de la section 6.1).

[6] Remarquer que les attributs (quand ils sont considérés) sont aussi des nœuds associés à des valeurs et sont donc soumis à la même condition.

La correspondance avec des requêtes arbre est aussi illustrée.

Pour toutes les contraintes de la table 6.2.1, C est un chemin qui commence à la racine et représente le *contexte* dans lequel la contrainte doit être vérifiée. Si $C = \epsilon$, alors les chemins suivants partent de la racine et les contraintes peuvent être présentées de façon abrégée, sans le chemin C. Dans ce cas, nous parlons d'une *contrainte absolue*, alors que si $C \neq \epsilon$ nous avons une *contrainte relative*.

Nous supposons l'existence d'une fonction de traduction qui permet le passage d'une requête arbre aux sous-chemins nécessaires pour la définition d'une contrainte et vice-versa. Pour cela nous étendons la définition de requête arbre (définition 6.1.1) de façon à marquer des nœuds spéciaux, ayant un rôle important dans la définition des contraintes. Par exemple, nous devons ajouter à `Query` le nœud c, le nœud contexte, et nœud tg, le nœud cible (*target*) : $\texttt{Query} = (D_q, q, Lab_q, c, tg, S_q)$[7].

Dans une requête arbre, le dernier nœud du chemin C est donc marqué comme étant le nœud contexte de la requête. Pour les contraintes de clé et de clé étrangère, Tg est un chemin qui commence au nœud contexte pour ensuite déterminer le nœud cible, c'est-à-dire, celui qui sera identifié par les valeurs de la clé. Des conditions particulières sont imposées à un nœud cible lors de la définition des clés. Dans la suite nous définissons la sémantique de chacune de quatre contraintes.

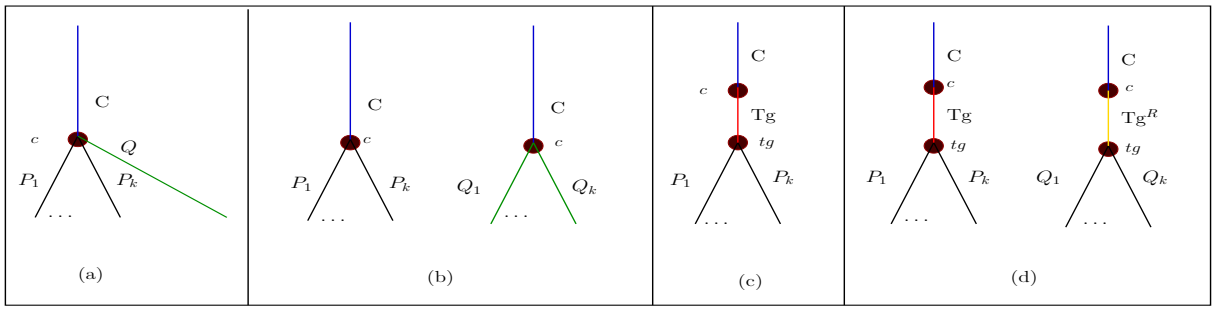


FIG. 6.3: Requêtes arbre pour la spécification des contraintes d'intégrité : (a) XFD, (b) XIC , (c) XKey et (d) XForeignKey.

6.2.1 Dépendances fonctionnelles

Nous reformulons la définition de la sémantique des XFD en utilisant les concepts utilisés dans ce mémoire. Tout d'abord nous considérons le cas où les informations sont complètes.

Définition 6.2.1 - Satisfaction d'une XFD - Soient $T = (D, t)$ un arbre XML et `Query` la requête arbre de la figure 6.3(a), correspondant à la XFD $\gamma = (C, (\{P_1\ [E_1], \ldots, P_k\ [E_k]\} \rightarrow Q\ [E]))$. L'arbre T satisfait γ (et cela s'écrit $T \models \gamma$) si quels que soient les parcours arborescents $\Pi^1_{\texttt{Query}}(T)$ et $\Pi^2_{\texttt{Query}}(T)$, obtenus par des projections de T sur `Query` coïncidant au moins sur C, la condition suivante est respectée :

$$\text{Si } \tau^1_{\texttt{Query}}[C/P_1, \ldots, C/P_k] =_{E_i, i\in[1\ldots k]} \tau^2_{\texttt{Query}}[C/P_1, \ldots, C/P_k] \text{ alors } \tau^1_{\texttt{Query}}[C/Q] =_E \tau^2_{\texttt{Query}}[C/Q]$$

où $\tau^1_{\texttt{Query}}$, $\tau^2_{\texttt{Query}}$, sont des n-uplets résultats de `Query` sur T, par rapport aux projections $\Pi^1_{\texttt{Query}}(T)$ et $\Pi^2_{\texttt{Query}}(T)$, respectivement et E_i indique un type d'égalité. □

Exemple 6.2.1 Considérons les XFD suivantes sur l'arbre de la figure 6.1 :

XFD_1 : $(fac, (\{/students/student/id\} \rightarrow /students/student\ [N]\ ))$: Deux étudiants distincts (c'est-à-dire, représentés par deux nœuds distincts) ne peuvent pas avoir le même numéro d'identification. Le contexte de XFD_1 est la racine et donc la dépendance doit être vérifiée dans tout l'arbre T.

XFD_2 : $(fac, (\{/students/student/id\} \rightarrow /students/student))$: Deux étudiants ayant le même numéro d'identification doivent être égaux en valeur, dans tout l'arbre T.

XFD_3 : $(fac/students, (\{/student/id\} \rightarrow /student\ [N]\ ))$: Cette dépendance est similaire à XFD_1, mais son contexte n'est pas le même. Dans le contexte *students*, deux étudiants distincts ne peuvent pas

[7] D'après la définition 6.1.3, la position contexte sur un arbre T est définie par $h(c)$ alors que la position cible est définie par $h(tg)$. Dans la section 6.3 nous montrons que notre algorithme de validation de contraintes trouve ces positions en utilisant des automates d'état finis.

avoir le même numéro d'identification.

$XFD_4 : (fac, (\{/course/name, /course/fromDept\} \rightarrow /course/code))$: Les cours d'un même département ayant le même titre, ont le même code. □

Dans le cas d'informations incomplètes, comme pour le modèle relationnel, deux types de satisfaction des XFD sont envisagées. Une *satisfaction forte* suppose que la XFD doit être satisfaite dans tous les mondes possibles. Cela veut dire qu'un document incomplet est valide si seulement si le remplacement d'une valeur *null* par une valeur différente de *null* n'implique pas une revalidation par rapport à la XFD. Une *satisfaction faible* suppose que la XFD doit être satisfaite dans au moins un monde : dans ce cas, si une valeur *null* est remplacée par une valeur différente de *null*, la revalidation est nécessaire pour vérifier si la XFD est toujours respectée.

La satisfaction faible des XFD peut être traitée comme dans le modèle relationnel. Pour cela, au moment de construire les n-uplets résultats de `Query` sur T, nous considérons que les valeurs *null* sont différentes les unes des autres. Une fois que nous avons les n-uplets τ, nous appliquons l'algorithme appelé *chase* [LL99]. Le principe de la procédure *chase* consiste à exécuter la boucle suivante, dont nous expliquons les concepts dans la suite. Soit R une instance de relation composée de tous les n-uplets $\tau[P_1, \ldots, P_n, Q]$, qui contiennent éventuellement des valeurs inconnues $\perp_{i_j}$.

Tant que il existe $\tau_1, \tau_2 \in R$ et
il existe une XFD $P \rightarrow Q$ telle que
$\tau_1[P] = \tau_2[P]$ et $\neg(\tau_1[Q] \cong \tau_2[Q])$ **faire**
$\tau_1[Q], \tau_2[Q] := lub(\tau_1[Q], \tau_2[Q])$

Dans cet algorithme, les concepts de base sont les suivants :

1. On considère un domaine **EV** contenant les valeurs (ou données) possibles dans T, étendu par des valeurs *null* (qui peuvent être distinguées ou non, selon la politique utilisée pour traiter les *null*).
2. Étant données deux valeurs ν_i et ν_j dans **EV**, ν_i est équivalent en information (*information-wise equivalent*) à ν_j ($\nu_i \cong \nu_j$), ssi ν_i et ν_j sont identiques syntaxiquement, c'est-à-dire qu'ils ont les mêmes noms. Si ν_i n'est pas équivalent en information à ν_j, on écrit $\neg(\nu_i \cong \nu_j)$. Par exemple, $\perp \cong \perp$, $\perp_2 \cong \perp_2$ et $iris \cong iris$, mais $\neg(iris \cong hilary)$, $\neg(\perp_2 \cong \perp_1)$.
3. L'égalité utilisée dans le *chase* est une égalité à trois valeurs (vraie, fausse et inconnue). Étant données deux valeurs ν_i et ν_j dans **EV**, l'égalité à trois valeurs est définie selon les cas ci-dessous :
   (a) Si ν_i et ν_j sont des valeurs différentes de *null* alors $\nu_i = \nu_j$ est évaluée comme vraie si $\nu_i \cong \nu_j$; sinon elle est évaluée comme fausse.
   (b) $\perp = \perp$ est évaluée comme inconnue.
   (c) $\perp = \perp_i$ est évaluée comme inconnue.
   (d) $\perp_j = \perp_i$ est évaluée comme vraie si $i = j$, sinon elle est évaluée comme inconnue.
   (e) Si ν_i est une valeur *null* (avec ou sans indices) et ν_j est une valeur différente de *null*, alors $\nu_i = \nu_j$ est évaluée comme inconnue.
4. Une relation d'ordre partiel ($\sqsubseteq$) est définie sur les valeurs. Les valeurs différentes de *null* sont incomparables. De plus nous avons $\perp \sqsubseteq nonnull\ values \sqsubseteq inc$ où *inc* est un type de valeur *null* qui indique une incohérence.
5. *Least upper bound operator* (lub) : la plus petite borne supérieure de deux valeurs ν_1 et ν_2 dans **EV** est définie de la façon suivante :
   (a) Si $\nu_1 \sqsubseteq \nu_2$ alors $lub(\nu_1, \nu_2) = \nu_2$; sinon
   (b) Si $\nu_2 \sqsubseteq \nu_1$ alors $lub(\nu_1, \nu_2) = \nu_1$; sinon
   (c) Si $\neg(\nu_2 \sqsubseteq \nu_1)$ et $\neg(\nu_1 \sqsubseteq \nu_2)$ alors $lub(\nu_1, \nu_2) = inc$.

L'instance de relation R satisfait faiblement une XFD ssi la procédure *chase* est cohérente (aucune valeur *inc* n'est introduite par la boucle). Ainsi, pour prendre en compte les informations incomplètes, il suffit de considérer une extension de la définition 6.2.1 ayant comme condition pour la satisfaction la réussite du chase ([BHL07]).

Exemple 6.2.2 Soit la XFD

$$\gamma = (fac, (\{//student/courseTaken/courseCode\} \rightarrow //student/courseTaken/deptName))$$

exprimée par une requête arbre `Query`. La relation R ci-dessous obtenue à partir des projections $\Pi_{\texttt{Query}}(T)$ est la même que celle illustrée dans l'exemple 6.1.7 où les valeurs *null* sont considérées comme étant différentes.
Remarquer que $P_1 = fac//student/courseTaken/courseCode/$ et $Q = fac//student/courseTaken/deptName/$

P_1	Q
C-DB-07S2	Computing
$\perp_1$	Maths
C-Theory-07S1	Computing
C-Theory-07S1	$\perp_2$
C-Prog-07S2	Maths
C-Logic-07S2	$\perp_3$

Comme la table obtenue (ci-après) en utilisant le *chase* ne contient pas des incohérences, notre document XML satisfait faiblement la XFD γ.

P_1	Q
C-DB-07S2	Computing
$\perp_1$	Maths
C-Theory-07S1	Computing
C-Theory-07S1	**Computing**
C-Prog-07S2	Maths
C-Logic-07S2	$\perp_3$

□

Nous référençons [AL02, HT06, VLL04] pour des propositions d'axiomatisation ou des formes normales. Il est aussi possible de choisir une satisfaction forte des XFD. Dans ce cas, il faut considérer toutes les valeurs *null* comme $\perp$ (sans différence de nom) pour construire les n-uplets qui composent R et d'appliquer l'algorithme *SatFort* sur R ([LL99]).

6.2.2 Dépendances d'inclusion

Comme dans la section précédente, nous reformulons la définition de la sémantique des XID.

Définition 6.2.2 - Satisfaction d'une XID : Soient $T = (D, t)$ un arbre XML et $\texttt{Query}_P$, $\texttt{Query}_Q$ les requêtes arbre de la figure 6.3(b), correspondant à la XID $\gamma = (C, (\{P_1, \ldots, P_k\} \subseteq \{Q_1, \ldots, Q_k\}))$. L'arbre T satisfait γ (et cela s'écrit $T \models \gamma$) si pour tout parcours arborescent défini par $\Pi_{\texttt{Query}_P}(T)$, il existe un parcours défini par $\Pi_{\texttt{Query}_Q}(T)$, qui coïncide avec le parcours arborescent de $\Pi_{\texttt{Query}_P}(T)$ au moins en C, et tel que :

$$\tau_{\texttt{Query}_P}[C/P_1, \ldots, C/P_k] \sqsubseteq \tau_{\texttt{Query}_Q}[C/Q_1, \ldots, C/Q_k]$$

où $\tau_{\texttt{Query}_Q}$, $\tau_{\texttt{Query}_P}$ sont des n-uplets résultats de $\texttt{Query}_Q$ et $\texttt{Query}_P$ sur T par rapport aux projections $\Pi_{\texttt{Query}_Q}(T)$ et $\Pi_{\texttt{Query}_P}(T)$ respectivement, et où le symbole $\sqsubseteq$ indique la relation d'ordre partiel introduite dans la section 6.2.1 ($\perp \sqsubseteq$ *nonnull values* $\sqsubseteq$ *inc*) et prend en compte l'égalité multivaluée. □

Exemple 6.2.3 Soit l'arbre T de la figure 6.1. Soit la XID

$$\gamma = (fac, (\{//student/courseTaken/courseCode\} \subseteq \{/course/code\})$$

exprimée par les requêtes arbres $\texttt{Query}_1$ et $\texttt{Query}_2$. À titre d'exemple, pour illustrer la définition 6.2.2, nous considérons que les n-uples obtenus via deux projections $\Pi_{\texttt{Query}_1}(T)$ sont stockés dans la relation $R_{\texttt{Query}_1} = \{\langle C{-}DB{-}07S2\rangle, \langle \perp_1\rangle\}$. De manière similaire, nous obtenons $R_{\texttt{Query}_2} = \{\langle C{-}DB{-}07S2\rangle, \langle C{-}Theory{-}07S1\rangle\}$ via des projections $\Pi_{\texttt{Query}_2}(T)$. Dans ce cas, pour ces projections, la XID est satisfaite. □

6.2.3 Clés

Selon la syntaxe exposée dans la table 6.2.1 les clés sont composées de trois parties : (*i*) le contexte dans lequel la clé se vérifie, (*ii*) l'élément identifié par la clé, et (*iii*) les parties de l'élément qui forment la

clé. La sémantique d'une clé est donnée par la définition ci-dessous : cette définition impose qu'à partir de chaque nœud n identifié par la clé (c'est-à-dire, chaque nœud target) il existe une et une seule instance pour chaque chemin P_i (troisième partie d'une clé).

Définition 6.2.3 - Satisfaction d'une clé : Soient $T = (D, t)$ un arbre XML et `Query` la requête arbre de la figure 6.3(c), correspondant à la clé $\gamma = (C, (Tg, \{P_1, \ldots, P_k\}))$. Quelque soient les projections $\Pi^1_{\texttt{Query}}(T)$ et $\Pi^2_{\texttt{Query}}(T)$ coïncidant au moins en C, les propriétés suivantes sont respectées :

1. Soit I^i l'instance du chemin C/Tg sur T et préfixe commun des parcours simples dans la projection $\Pi^i_{\texttt{Query}}(T)$. À partir de la position $LastPosition(I^i)$, il n'existe pas d'autre projection possible de P_j (pour tout $1 \leq j \leq k$) que celle de $\Pi^i_{\texttt{Query}}(T)$.
2. Le n-uplet $\tau^1_{\texttt{Query}}[C/Tg/P_1, \ldots, C/Tg/P_k]$ ne contient aucune valeur *null* et si $\tau^1_{\texttt{Query}}[C/Tg/P_1 \ldots C/Tg/P_k] = \tau^2_{\texttt{Query}}[C/Tg/P_1 \ldots C/Tg/P_k]$ alors $\tau^1_{\texttt{Query}}[C/Tg] =_N \tau^2_{\texttt{Query}}[C/Tg]$.

où $\tau^1_{\texttt{Query}}$, $\tau^2_{\texttt{Query}}$ sont des n-uplets résultats de `Query` sur T par rapport aux projections $\Pi^1_{\texttt{Query}}(T)$ et $\Pi^2_{\texttt{Query}}(T)$, respectivement. □

Sur le document de la figure 6.1, soit la clé $K_1 = (fac(/students/student, (\{id\})))$ qui, dans notre exemple, indique que le numéro d'identification d'un étudiant est unique dans tout le document (puisque la racine est le nœud *fac*).

6.2.4 Clés étrangères

Les clés étrangères sont un cas particulier de dépendances d'inclusion où le coté droit de la dépendance est une clé. Un arbre T satisfait une clé étrangère $\gamma = (C, (Tg^R, \{P^R_1, \ldots, P^R_k\}) \subseteq (Tg, \{P_1, \ldots, P_k\}))$ ssi la clé $(C, (Tg, \{P_1, \ldots, P_k\}))$ est satisfaite et si la XID exprimée par la figure 6.3(d) l'est aussi (définition 6.2.2).

Comme exemple sur le document de la figure 6.1, soit la clé étrangère FK_1 :

$$(fac, (/students/student, \{/courseTaken/courseCode\} \subseteq (/students/course, \{code\}))$$

qui assure que les étudiants suivent des cours enregistrés à la fac.

6.3 Vérification des contraintes d'intégrité

Comme dans la validation de schéma, présentée dans le chapitre 4, dans nos travaux nous proposons une validation des contraintes d'intégrité qui correspond à un parcours à la SAX de l'arbre XML. La première validation (*from scratch*) peut stocker certaines informations qui seront utilisées et mises à jour pendant la validation incrémentale. Dans nos travaux [BCH+07, BHL07], nous avons considéré seulement la validation par rapport aux clés et aux clés étrangères. Néanmoins, cette méthode peut être adaptée pour les autres contraintes d'intégrité. Dans ce mémoire nous proposons une description générale de l'étape de validation, par rapport aux contraintes d'intégrité de la section précédente.

La syntaxe de nos contraintes d'intégrité inspire une méthode de validation homogène qui nécessite simplement un paramétrage différent selon le type de contrainte que nous voulons vérifier. Cette méthode de validation est composée de deux parties principales, à savoir :

1. Déterminer les nœuds concernés par la contrainte : les nœuds *contextes* et les nœuds *finaux*, comportant les informations qui doivent être comparées. Dans le cas des clés et des clés étrangères il faut aussi déterminer les nœuds *cibles*.
2. Remonter les valeurs impliquées dans la contrainte vers leur contexte afin de les tester à ce niveau.

Comme nous avons vu dans la section 6.2, chaque contrainte est spécifiée par différentes parties d'un chemin simple. Les nœuds sélectionnés via ces différentes parties de chemin ont un rôle bien défini dans la vérification de la contrainte. Pour déterminer les nœuds concernés par une contrainte, nous considérons ces différentes parties de chemin la composant et nous utilisons un automate d'états fini pour chacune. En d'autres termes, étant donné un chemin défini par deux parties principales $P = C/Q$, au lieu de travailler

avec un seul automate d'états fini A_P, nous utilisons les deux automates A_C et A_Q pour pouvoir bien établir le rôle des nœuds importants dans le contexte de la contrainte.

La table 6.3.1 indique les automates d'états finis (FSA) nécessaires pour chacune de nos contraintes. Les automates A_P et A_Q sont spéciaux. Les multiples chemins $(P_1, \ldots, P_k)$ sont représentés dans un automate A_P (via des branches OR) mais son comportement correspond à considérer plusieurs automates, un pour chaque chemin P_i. Par exemple, la contraint $(m, \{a/b, c/d\}$ définit $P_1 = a/b$ et $P_2 = c/d$. Pour simplier les explications, l'automate A_P est représenté par un diagramme ayant une bifurcation partant d'un état initial et allant dans deux états différents, selon la lecture de a ou c. Néanmoins, le comportement de A_P consiste à considérer deux automates, à savoir : un construit à partir de a/b et l'autre construit à partir de c/d. L'automate A_Q est construit de façon similaire. De plus, remarquer que, selon la contrainte, nous pouvons avoir différents automates pour représenter les différentes parties des chemins. Pour les dépendances fonctionnelles et d'inclusion, l'automate A_P représente le coté gauche et l'automate A_Q représente le coté droit. Les clés et les clés étrangères ont besoin des automates A_{Tg}.

La procédure de vérification des contraintes d'intégrité peut être effectuée par une grammaire d'attributs. Les grammaires d'attributs (introduites par Knutt en 1968) fournissent un mécanisme pour marquer les nœuds d'un arbre avec des *attributs* via des règles sémantiques qui peuvent fonctionner de façon *bottom-up* (pour les attributs synthétisés) ou de façon *top-down* (pour les attributs hérités) [ASU88]. Ainsi, une grammaire d'attributs est une grammaire hors contexte, étendue par des attributs. Pour ajouter de l'information aux symboles non terminaux d'une grammaire hors contexte nous leur associons un ensemble d'attributs. Chaque attribut a un domaine de valeurs. La valeur de chaque attribut à chaque nœud de l'arbre est définie par des règles sémantiques.

Table 6.3.1 - *Contraintes d'intégrité : leurs automates d'états finis et leurs attributs*

Contraintes	**FSA**	**Types d'attributs**
$(C, (\{P_1\ [E_1], \ldots, P_k\ [E_k]\}\ \rightarrow Q\ [E]))$	A_C, A_P et A_Q	hérité : $conf$ synthétisés : c et f
$(C, (\{P_1, \ldots, P_k\} \subseteq \{Q_1, \ldots, Q_k\}))$	A_C, A_P et A_Q	hérité : $conf$ synthétisés : c et f
$(C, (Tg, \{P_1, \ldots, P_k\}))$	A_C, A_{Tg} et A_P	hérité : $conf$ synthétisés : c, tg et f
$(C, (Tg^R, \{P_1^R, \ldots, P_k^R\}) \subseteq (Tg, \{P_1, \ldots, P_k\}))$	A_C, A_{Tg}^R, A_P^R A_{Tg} et A_P	hérité : $conf$ synthétisés : c, tg et f

Dans ce cadre de vérification des contraintes, la grammaire à enrichir avec des attributs aurait pu être celle qui définit le type du document, c'est-à-dire, le schéma (section 2.2). Néanmoins, dans nos travaux, comme les contraintes d'intégrité sont des contraintes indépendantes du schéma, la grammaire utilisée est celle qui décrit un arbre XML général, à savoir :
(1) Une règle générale pour indiquer, à partir d'un nœud ayant l'étiquette a, la dérivation de n fils : $A \rightarrow \alpha_1, \ldots, \alpha_n$
(2) Une règle spéciale pour définir la racine : $R \rightarrow \alpha_1, \ldots, \alpha_n$.
(3) Une règle spéciale pour définir des feuilles associées à des données : $A \rightarrow data$.
Cette grammaire est étendue par l'ajout de règles sémantiques composées des attributs et des actions contenant des informations concernant les contraintes d'intégrité.

Pendant le parcours de la racine vers les feuilles l'algorithme de validation détermine, avec l'aide des automates à états finis, le rôle de chaque nœud par rapport à la contrainte d'intégrité : chaque nœud contient un attribut, appelé $conf_i$, pour chaque contrainte γ_i devant être vérifiée. *L'attribut hérité* $conf_i$ contient une information (nous parlons de la *configuration de l'automate d'états fini*) qui exprime le rôle du nœud par rapport à la contrainte γ_i.

Ainsi, nous parcourons l'arbre XML en suivant les automates d'état finis correspondant aux contraintes. La valeur de l'attribut hérité $conf_i$ reflète le rôle d'un nœud par rapport à la contrainte γ_i. Soit A_C l'automate d'états fini concernant le chemin contexte de γ_i. Nous commençons avec A_C dans son état initial e_0 : $conf_i$ de la racine contient la configuration $A_C.e_0$ (c'est-à-dire, $conf_i = \{A_C.e_0\}$). Ensuite, nous calculons $conf_i$ pour chaque fils de la racine en exécutant l'automate A_C sans oublier de vérifier si nous avons besoin de changer d'automate, c'est-à-dire, si nous sommes arrivés à un état final de A_C. Quand cela se produit, l'automate suivant (A_{Tg} ou A_P) est utilisé dans le calcul de $conf_i$. La procédure continue

et ainsi les nœuds contexte ont pour configuration $A_C.f_{A_C.}$ où $f_{A_C.}$ est l'état final de A_C, les nœuds cible (pour les clés) ont pour configuration $A_{Tg}.f_{A_{Tg}}$ où $f_{A_{Tg}}$ est l'état final de A_{Tg} et les nœuds finaux ont pour configuration $A_P.f_{A_P}$ où f_{A_P} est l'état final de A_P. De plus, les noeuds se trouvant sur les chemins ont une configuration associée, ce qui va permettre de faire transiter les valeurs à tester depuis les feuilles vers les nœuds contextes.

Exemple 6.3.1 Dans la figure 6.4 nous faisons un *zoom* sur une partie du document de la figure 6.1 pour illustrer la validation de la dépendance fonctionnelle :

$\gamma = (fac/students, (\{/student/courseTaken/courseCode\} \rightarrow \{/student/courseTaken/deptName\}))$

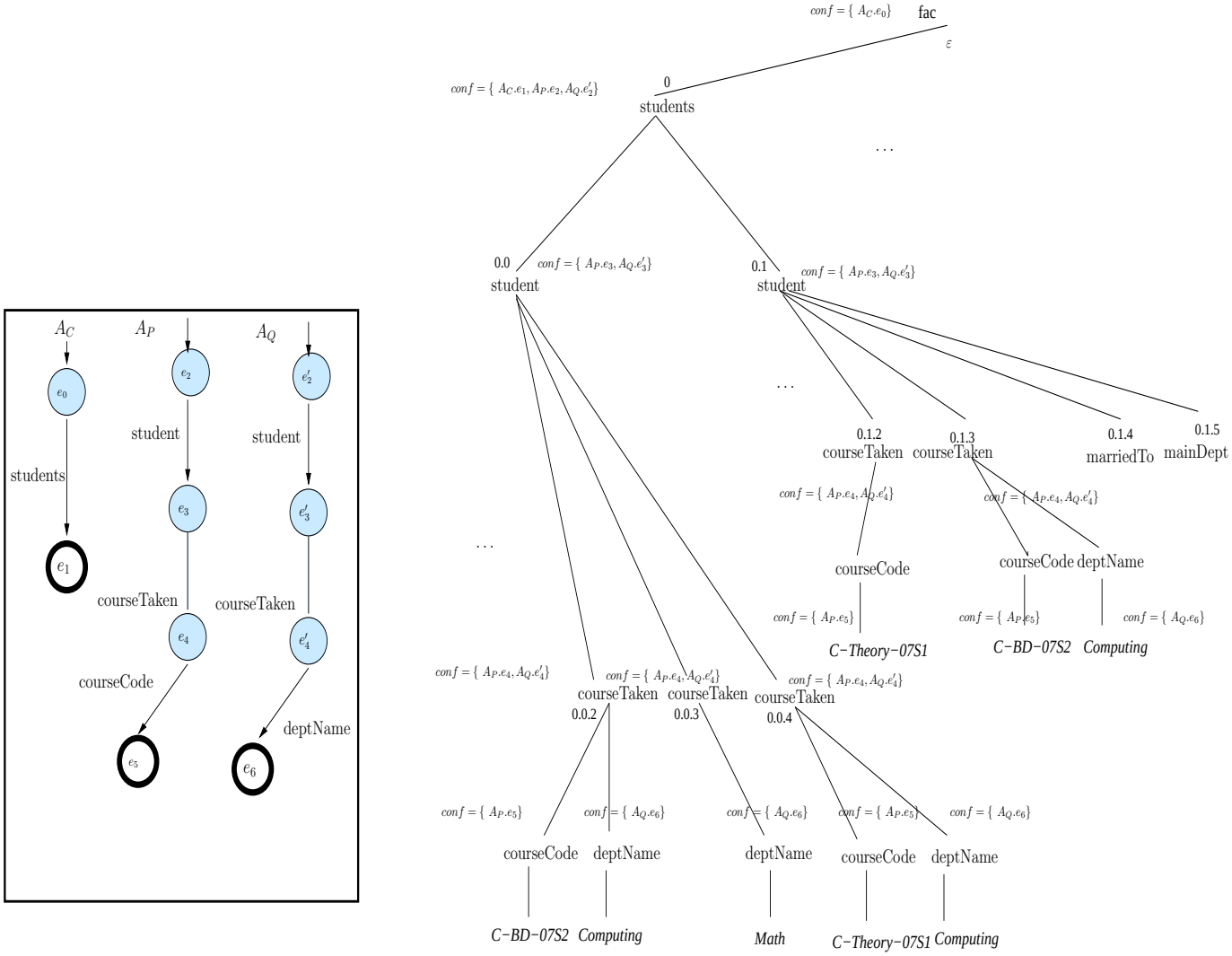


FIG. 6.4: Idée générale de la validation de la XFD $\gamma = (fac/students, (\{/student/courseTaken/courseCode\} \rightarrow \{/student/courseTaken/deptName\}))$. Le calcul des attributs hérités.

Les automates A_C, A_P et A_Q utilisés dans la validation sont aussi illustrés dans la figure 6.4. L'attribut $conf$ est calculé de manière *top-down*. Il est initialisé avec la configuration $A_C.e_0$ pour indiquer que le point de départ est l'état initial de l'automate A_C. À la position 0 de l'arbre, $conf = \{A_C.e_1, A_P.e_2, A_Q.e'_2\}$, nous sommes à l'état final de A_C et à l'état initial[8] de A_P et A_Q . □

Une fois arrivé aux feuilles, nous utilisons la direction *bottom-up*, c'est-à-dire, les attributs synthétisés pour remonter des valeurs concernant les contraintes à vérifier (voir la figure 6.5). Pour chaque contrainte, différents attributs sont utilisés : chaque rôle différent, initialement spécifié par la syntaxe des contraintes, correspond à un attribut hérité. La table 6.3.1 donne un aperçu des attributs nécessaires à chaque contrainte. Les dépendances fonctionnelles et d'inclusion ont besoin des attributs c pour les

[8]Dans l'implémentation ce n'est pas nécessaire de stocker l'état final de l'automate A_C

valeurs contexte et f pour les valeurs finales. Les clés ont aussi un attribut tg pour les valeurs cibles. À chaque nœud ces attributs reçoivent des valeurs qui varient selon leur rôle par rapport à la contrainte γ_i et aussi selon les attributs hérités de leurs fils.

Aux feuilles associées à une contrainte, les attributs f récupèrent les valeurs qui doivent être remontées. Aux positions p, correspondant à un ancêtres déterminé par la contrainte ces valeurs sont regroupés en n-uplet. Ensuite, les n-uplets sont regroupés en ensemble de n-uplets jusqu'à arriver au contexte pour une vérification qui varie selon la contrainte prise en compte. L'attribut c stocke le résultat de cette vérification. Les résultats des vérifications sont remontés à la racine, donnant la décision pour le document complet. À chaque étape de cette remontée, des tests spécifiques aux contraintes peuvent être faits, comme par exemple tester un n-uplet sur un nœud target pour vérifier si toutes les composantes d'une clé existent (c'est-à-dire, si tous les chemins $P_1 \dots P_k$ sont associés à des instances de chemins complètes).

Remarque : Dans ce mémoire nous présentons une idée générale de la méthode. Les détails de l'approche, exposées dans [Lim07], ne sont pas traités ici. En effet, pour les XFD, [Lim07] utilise un attribut f pour chaque chemin $P_1, \dots, P_k, Q$ (c'est-à-dire, $f_1, \dots, f_k, f_Q$). De plus, la combinaisons des valeurs pour construire les n-uplets contenant les données obtenus via $P_1, \dots, P_k$ ainsi que les données obtenus via le chemin Q est faite via un attribut $f_{combination}$. Ici, nous illustrons cette méthode via un exemple très simple et sans faire la distinction entre les différents types d'attributs f. Nous remarquons seulement que la combinaison des valeurs est faite sur les nœuds correspondant aux intersections entre les instances de chemins de la XFD.

Exemple 6.3.2 Dans la figure 6.5 nous considérons encore le *zoom* de la figure 6.4 pour illustrer le calcul des attributs c et f dans la validation de la dépendance fonctionnelle :

$$\gamma = (fac/students, (\{/student/courseTaken/courseCode\} \rightarrow \{/student/courseTaken/deptName\}))$$

Les attributs c et f sont calculés de façon *bottom-up*. Toutes les valeurs concernant γ sont stockées aux feuilles, dans les attributs f pour ensuite être regroupées en n-uplets (dans la figure 6.5, les positions père de celles contenant les valeurs). Dans la figure 6.5, nous remarquons qu'aux positions étiquetées par *student* des ensembles de n-uplets sont formés et que, aux positions étiquetées *students*, les tests concernant la XFD γ sont effectués. Le résultat montre (même si l'exemple ne montre pas l'instance complète du document) que l'arbre XML est valide par rapport à γ. □

En conclusion, les *attributs synthétisés* servent à ce processus de remontée : un attribut f est utilisé entre une feuille et un nœud cible (dans le cas des clés) ou entre une feuille et un nœud contexte (dans le cas des dépendances) ; un attribut tg est utilisé entre un nœud cible et un nœud contexte, et un attribut c est utilisé entre un nœud contexte et la racine. L'attribut c est donc calculé à partir des attributs précédents (tg ou f, selon la contrainte) et l'attribut tg est calculé à partir de l'attribut f.

Une vérification d'une contrainte d'intégrité se réalise donc pendant le parcours linéaire du document XML (*à la SAX*). Pendant ce parcours, nous construisons un index des valeurs des contraintes, utile pour vérifier les dépendances d'inclusion et les clés étrangères et pour réaliser une vérification incrémentale en cas de mise à jour du document. Dans [BCH+07], nous discutons les détails de toute la méthode de validation décrite ci-dessus appliquée aux clés et aux clés étrangères.

6.4 Validation incrémentale par rapport aux contraintes d'intégrité

Étant donné un arbre $T = (D, t)$, un ensemble de contraintes d'intégrité et une liste de mises à jour L, la validation incrémentale par rapport aux contraintes d'intégrité consiste à vérifier si l'arbre mis à jour respecte ces contraintes en testant seulement les parties concernées par les mises à jour. Nous proposons un algorithme similaire à l'algorithme 4.1.1 qui fait un parcours à la SAX de l'arbre XML et qui traite seulement les positions suivantes[9] :

1. Toutes les positions qui sont un préfixe d'une position d'insertion.
2. Toutes les positions dont le préfixe est une position de suppression ou de remplacement.

[9]Remarquer que ce traitement consiste à recalculer les attributs des positions en question.

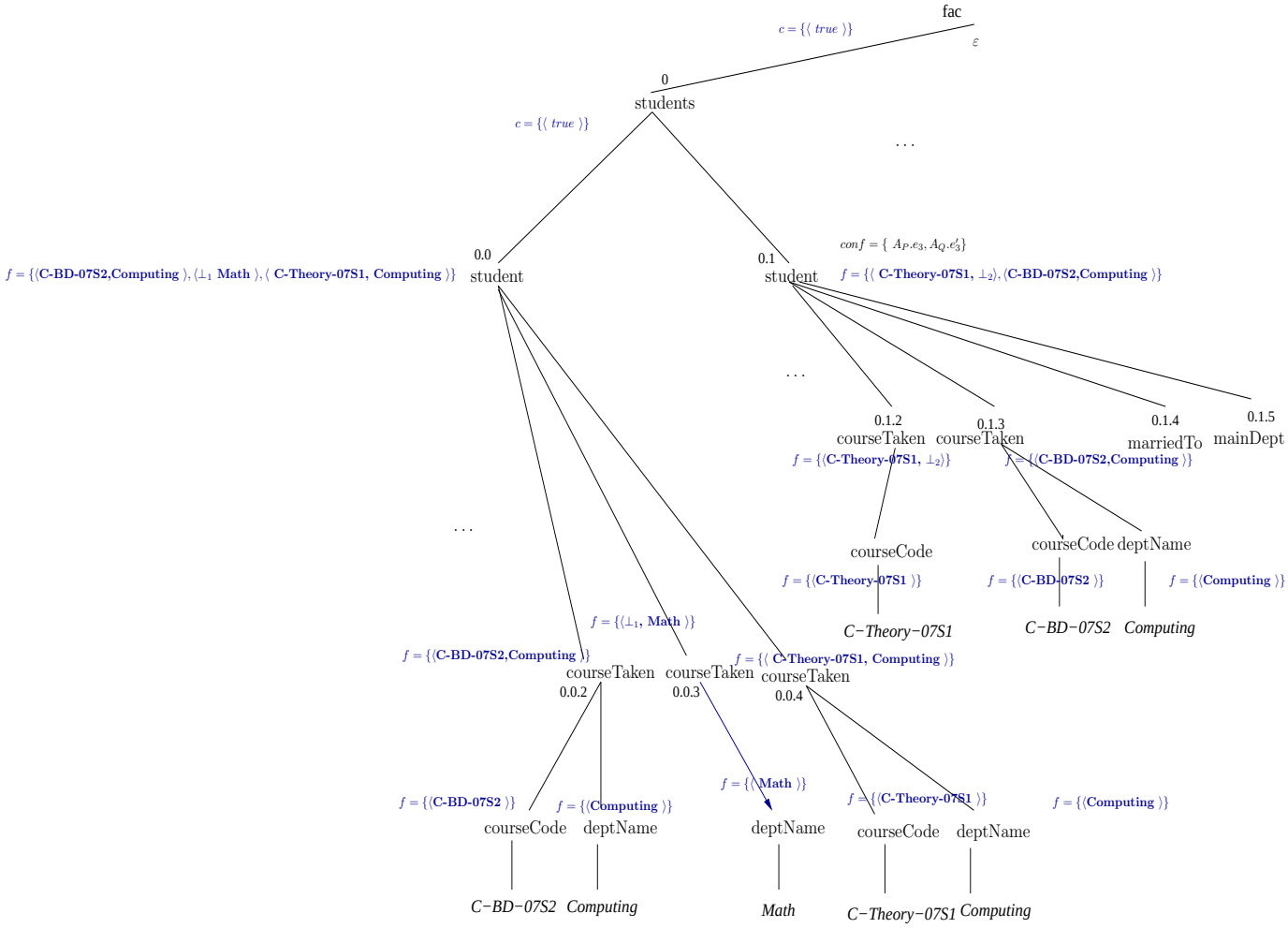


FIG. 6.5: Idée générale de la validation de la XFD $\gamma = (fac/students, (\{/student/courseTaken/courseCode\} \rightarrow \{/student/courseTaken/deptName\}))$. Le calcul des attributs synthétisés.

Pour effectuer une validation incrémentale par rapport aux contraintes d'intégrité, des structures auxiliaires stockant des valeurs associées aux contraintes sont nécessaires. Ces structures (une par contrainte) sont initialisées pendant la première validation *from scratch* et mises à jour pendant la validation incrémentale. L'algorithme suivant décrit notre méthode générale pour la validation incrémentale par rapport aux contraintes d'intégrité.

Algorithme 6.4.1 - Incremental Validation of Integrity Constraints wrt Multiple Updates

Input :
(*i*) *doc* : An XML document.
(*ii*) *UpdateTable* : A relation that contains updates to be performed on *doc*, similar to the one in Algorithm 4.1.1.
(*iii*) *ConstraintFSA* : Set of finite state automata describing the paths that appear in the constraints.
(*iv*) *AUX* : Structure that contains the values resulting from the last validation performed on *doc*.

Output :
If *doc* remains valid after all operations in *UpdateTable* the algorithm returns the boolean value *true*, otherwise *false*.

Local Variables :
(*i*) *CONF* : structure storing the inherited attributes.
(*ii*) *SYNT* : structure storing the synthesized attributes.
(*iii*) *AuxTemp* : copy of the initial *AUX*.

(1) $AuxTemp := AUX$
(2) $CONF_\epsilon := InitializeInhAttributes(ConstraintFSA)$
(3) **for each** *event v in doc* **do**
(4) **switch** (v) **do**
(5) **case** *start of element a at position p*
(6) Compute $CONF_p$ using $ConstraintFSA$
(7) **for each** `insert` operation on position p in $UpdateTable$ **do**
(8) **if** the insertion is not accepted **then** report "invalid" and halt
(9) **if** in $UpdateTable$ there is no update on a position p' *such that* $p \prec p'$
(10) **then** $skipSubTree(doc, a, p)$;
(11) **case** *end of element a at position p*
(12) Compute $SYNT_p$ using $CONF_p$
(13) **if** there is a `delete` operation on position p in $UpdateTable$ **do**
(14) **then if** the deletion is not accepted **then** report "invalid" and halt
(15) **if** there is a `replace` operation on position p in $UpdateTable$ **do**
(16) **then if** the replacement is not accepted **then** report "invalid" and halt
(17) **case** *value*
(18) $str := value(p)$
(19) Compute $SYNT_p$ using $CONF_p$ and str
(20) **if** $(\neg valid(AuxTemp))$ **then** report "invalid" and halt
(21) **return** *true* □

L'algorithme 6.4.1 commence par copier la structure AUX dans une "copie de travail" *AuxTemp*. Cette copie sera modifiée pendant la validation incrémentale qui prend en compte toute une liste de mises à jour. À la fin, quand toute la liste a été prise en compte, si *AuxTemp* est cohérente, la structure AUX sera remplacée par *AuxTemp*.

L'algorithme utilise deux structures pour stocker les valeurs des attributs, à savoir, $CONF$ et $SYNT$. À chaque position p, la structure $CONF_p$ stocke les différents rôles de p par rapport aux contraintes à vérifier. En effet, $CONF_p$ contient, pour la position p, un attribut $conf$ pour chaque contrainte. La structure $SYNT_p$ contient pour la position p, un n-uplet composé des attributs f, tg et c pour chaque contrainte.

En arrivant à une balise ouvrante à la position p, l'algorithme 6.4.1 calcule les valeurs héritées qui seront stockées dans $CONF_p$ et traite les insertions sur p. De façon similaire, en arrivant à une balise fermante à la position p, l'algorithme 6.4.1 calcule les valeurs synthétisées qui seront stockées dans $SYNT_p$ et traite les suppressions et les remplacements. Nous rappelons que le calcul des valeurs synthétisées dépend non seulement de $CONF_p$ mais aussi des résultats de chaque $SYNT_{p'}$, où p' est un fils de p.

Les insertions, les suppressions et les remplacements sont traités par des algorithmes spécifiques qui ne sont pas présentés dans ce mémoire ([BCH+07]). Ces algorithmes effectuent différents tests, selon la contrainte à vérifier. Ils sont implémentés comme des fonctions qui retournent *vrai* si les mises à jour sont acceptées et *faux* dans le cas contraire. Remarquer que nous restons dans le contexte des mises à jour multiples et donc, dans certaines circonstances, une mise à jour élémentaire peut être acceptée temporairement : la décision finale sera prise seulement après avoir pris en compte toute la liste de mises à jour. Par exemple, soit l'insertion d'une instance d'une clé étrangère à une position p et supposez que les valeurs de la clé correspondante n'existent pas dans l'arbre. Dans ce cas, nous pouvons accepter l'insertion en espérant que les valeurs de la clé correspondante seront insérées dans la suite de la mise à jour multiple. La structure AUX enregistre le fait que cette information n'est pas encore valide. Après avoir traité toute la liste, la structure AUX est vérifiée (ligne 20).

Dans [BCH+07] nous présentons la validation incrémentale des clés et clés étrangères. Nous référençons cet article (en annexe) pour les détails des algorithmes et de leur complexité ainsi que pour les résultats

```
<!DOCTYPE keyTree[
<!ELEMENT keyTree (context*)>
<!ATTLIST keyTree nameKey CDATA #REQUIRED>
<!ELEMENT context (target+)>
<!ATTLIST context pos CDATA #REQUIRED>
<!ELEMENT target (key+)>
<!ATTLIST target pos CDATA #REQUIRED refCount CDATA #REQUIRED>
<!ELEMENT key #PCDATA>]
```

FIG. 6.6: DTD décrivant la structure d'un *keyTree*

expérimentaux. Dans ce mémoire nous allons seulement illustrer la validation incrémentale via l'exemple qui est présenté dans la thèse de Adriana de Lima ([Lim07]).

Nous rappelons que pendant la validation *from scratch* par rapport aux clés et aux clés étrangères, la structure AUX (qui dans le cas des clé est appelée $KeyTree$) est crée. Le $KeyTree$ garde toutes les valeurs de la clé trouvées dans le document, indiquant la position du contexte correspondant, la position de la cible, et les valeurs associées à chaque composant de la clé. La figure 6.6 décrit cette structure d'index avec la notation de DTDs. Pour chaque clé K qui doit être respectée par le document XML, nous créons le $keyTree_K$ correspondant. Aussi, pour chaque clé, un compteur de références, appelé *refCount* stocke le nombre de clés étrangères faisant référence à la clé.

Exemple 6.4.1 Soient l'arbre T de la figure 6.7 et la séquence suivante de mises à jour :
(1) Une insertion d'une nouvelle *recette* à la position 0.1 ;
(2) Une suppression dans la position 0.2 ;
(3) Une suppression dans la position 0.3.0.

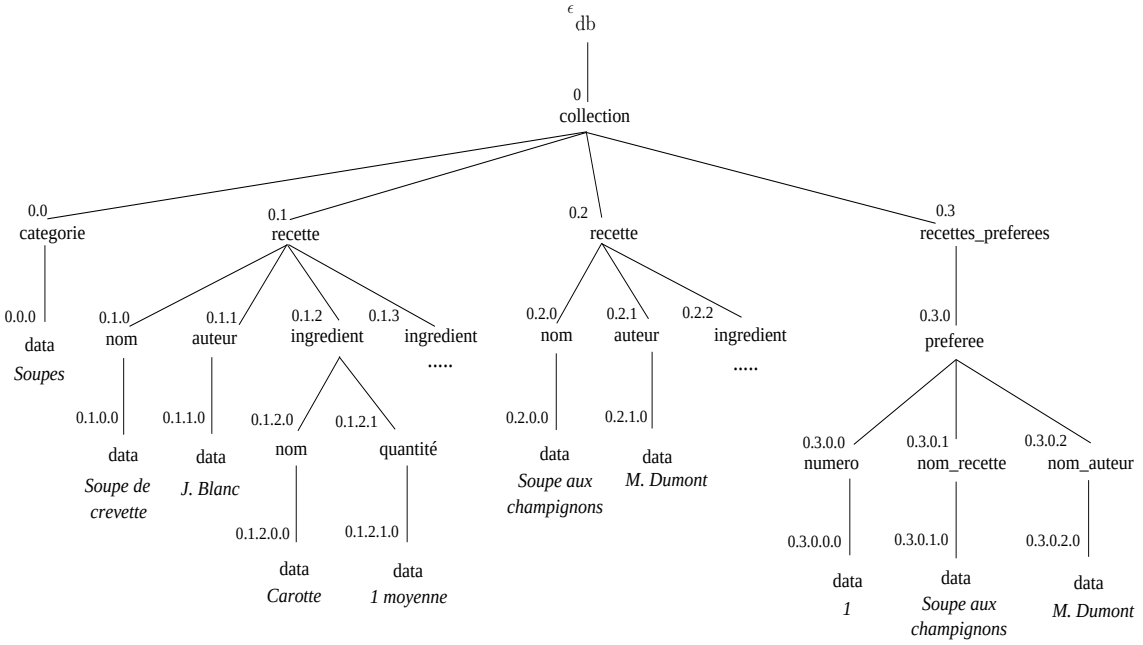


FIG. 6.7: Arbre XML qui décrit une collection de recettes.

L'arbre T initial est valide par rapport aux contraintes suivantes :

K : ($bd/collection$, ($//recette$, {$/nom$, $/auteur$}))
FK : ($bd/collection$, ($/recettes\_preferees/preferee$, {$/nom\_recette$, $/nom\_auteur$}) $\subseteq$ ($//recette$, {$/nom$, $/auteur$}))

Une copie du *keyTree* original est conservée dans $AuxTemp$ (ici appelé $keyTreeTemp$). La structure $keyTreeTemp$ est temporaire et sera employée dans la validation des mises à jour sur T concernant K et FK. La séquence de mises à jour est examinée pendant le parcours de l'arbre T, en suivant l'algorithme 6.4.1. Nous pouvons constater que les tests sont exécutés dans l'ordre suivant :

1. Quand la balise d'ouverture de l'élément *recette* dans la position 0.1 est atteinte, il existe une insertion devant être exécutée. Le nouveau sous-arbre représente une nouvelle recette qui contient des instructions et ingrédients pour la préparation d'une soupe de brocolis. Pour vérifier si l'insertion est valide, la procédure `insert` (ligne 7 de l'algorithme 6.4.1) est exécutée.

   Remarquer qu'une condition nécessaire pour l'insertion d'un nouveau sous-arbre T' dans un arbre T est que T' soit *valide localement*, c'est-à-dire, que T' n'ait aucun problème interne de validité par rapport aux contraintes qui sont posées sur T, comme par exemple, des valeurs doublées pour une clé. Ici, nous considérons l'insertion d'un arbre localement valide.

   En suivant l'algorithme `insert`, si la position de mise à jour est une position dans un chemin cible (comme c'est le cas ici), alors les nouvelles valeurs clés doivent être enregistrées dans $keyTreeTemp_K$ sous le contexte correspondant. Cependant, les valeurs clés peuvent exister déjà dans $keyTreeTemp_K$:
   (i) Si les valeurs clés déjà existantes sont venues d'une insertion incomplète (déclenchée par une insertion précédente d'une clé étrangère) alors $pos = 0$ et l'insertion est à compléter en remplaçant la valeur de cet attribut *pos* par une valeur de position d'insertion.
   (ii) Si la duplication ne vient pas d'une insertion incomplète, alors il existe des valeurs clés doublées et cette violation est marquée en ajoutant l'attribut *dup = "yes"* à l'instance de clé originale dans $keyTreeTemp_{K_i}$.

   Dans notre cas, les valeurs à insérer n'existent pas encore. Les nouvelles valeurs de clé sont ajoutées à $keyTreeTemp_K$ sous la collection de soupes (contexte à la position 0, préfixe de la position 0.1), comme le montre la figure 6.8. Notons que les attributs *pos* ne sont pas encore mis à jour.

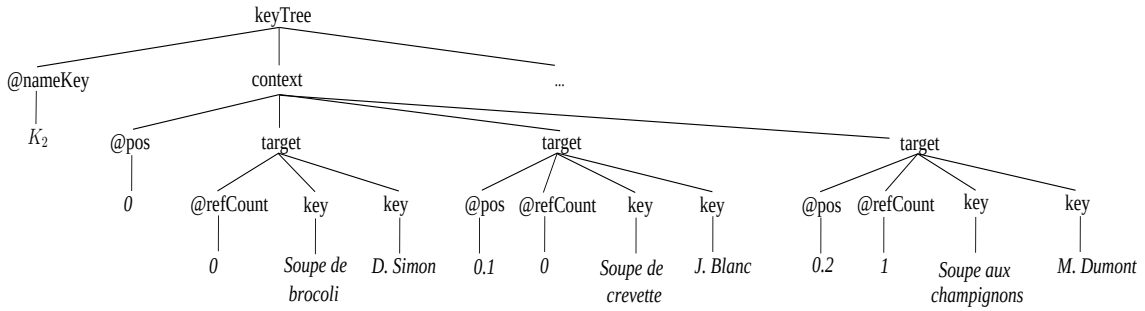


FIG. 6.8: $KeyTreeTemp_K$ après l'insertion dans la position 0.1.

2. Quand la balise de fermeture de l'élément *recette* dans la position 0.2 est atteinte, une suppression doit être exécutée (l'exclusion de la recette de *Soupe aux champignons*). Pour vérifier si la suppression est acceptée, l'opération delete (ligne 13 de l'algorithme 6.4.1) est exécutée.

   Si la position de suppression p est une position dans un chemin cible et s'il existe des instances de clé devant être supprimées sous un contexte de p, et si ces instances ne sont référencées par aucune clé étrangère, alors la suppression est acceptée. Sinon, le nœud cible correspondant à l'instance de clé dans $keyTreeTemp_K$ est marqué pour que la suppression soit faite postérieurement. La suppression sera faite à la fin si et seulement si toutes les références ont été retirées.

   Dans notre cas, la suppression concerne la clé K et donc les valeurs de K sous la position 0.2 doivent être exclues du $keyTreeTemp_K$ par rapport au contexte dans la position 0. La figure 6.9 montre que cette suppression est retardée, puisque l'attribut *refCount* associé à l'instance de clé *Soupe aux champignons* est égal à 1. Pour cela, l'attribut $del = yes$ est ajouté à l'instance de clé dont la position cible est 0.2.

3. Quand la balise de fermeture de l'élément *recette_preferee* dans la position 0.3.0 est atteinte, la suppression de la recette préférée ayant *numero = 1* doit être faite.

   Dans l'algorithme delete, si la position de mise à jour p est une position dans un chemin cible d'une clé étrangère, alors la suppression est acceptée et les attributs *refCount* correspondants sont décrémentés dans $keyTreeTemp_K$ pour les clés référencées. Si l'attribut *refCount* est affecté avec 0 et si l'instance de la clé dans $keyTreeTemp_{K_i}$ a été marquée précédemment pour être supprimée (attribut $del = yes$), alors cette instance est supprimée (et par conséquent, une violation temporaire a été corrigée).

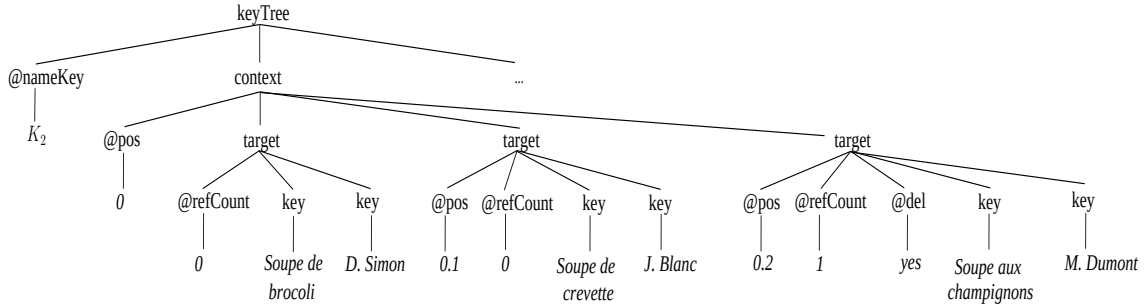


FIG. 6.9: $KeyTreeTemp_K$ après la suppression de la position 0.2.

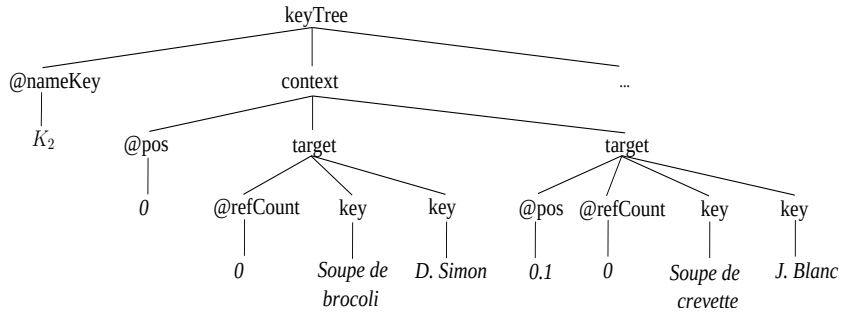


FIG. 6.10: $KeyTreeTemp_K$ après la suppression de la position 0.3.0.

Dans notre cas, la suppression concerne la clé étrangère FK (qui fait référence à K). Les valeurs au-dessous de la position 0.3.0 doivent être exclues de $keyTreeTemp_K$ dans la collection de soupes (contexte dans position 0). Cette exclusion est accomplie en retirant 1 à l'attribut *refCount* correspondant à *Soupe aux champignons*. À ce point, puisque *refCount* revient à 0 et le n-uplet associé à *Soupe aux champignons* a été marqué pour une suppression postérieure, alors cette suppression est effectuée, et la figure 6.10 montre le $keyTreeTemp_K$ résultant. À la fin de la séquence de mises à jour, il est nécessaire de traverser le $keyTreeTemp_K$ pour : (a) mettre à jour les attributs *pos* et (b) vérifier l'existence de marques de violation de contrainte. □

Dans l'algorithme 6.4.1, chaque position de mise à jour active une étape de validation qui dépend du type de la mise à jour demandée. Pour découvrir si une contrainte est concernée par une mise à jour, les algorithmes consultent la liste $List_{aux}$ qui contient le résultat de la validation locale du sous arbre T_p^{sub} (à insérer ou à supprimer). La liste $List_{aux}$ est une structure auxiliaire qui contient, pour le sous arbre T_p^{sub}, les valeurs stockées par les attributs synthétisés ainsi que certaines informations à stocker dans AUX. Elle est structurée selon les contraintes que nous voulons vérifier et est construite en temps $O(|T_p^{sub}|)$ car, en général, nous devons parcourir le sous arbre pour connaître les valeurs concernant les contraintes à valider.

Les algorithmes qui traitent les mises à jour font des tests sur $List_{aux}$ et comparent ses valeurs avec celles stockées dans AUX. Cette opération dépend du type de contrainte, mais peut être effectuée en temps $O(n.N)$ où n est le nombre de contraintes et N le nombre d'occurrences pour une contrainte. Ainsi, les m mises à jour sont traitées en temps $O(m(|T_p^{sub}| + n.N))$.

6.5 Travaux liés

Dans nos travaux nous nous sommes intéressés à la validation incrémentale des contraintes d'intégrité. Néanmoins, différents types de problèmes liés aux contraintes d'intégrité ont été traités dans la littérature. Dans cette section nous offrons un aperçu de ces travaux, en les classant dans différents thèmes. Nous considérons d'abord les différentes contraintes d'intégrité traitées en XML, en particulier les dépendances fonctionnelles. Comme dans le modèle relationnel, il est important d'étudier les problèmes de décision

concernant les contraintes d'intégrité. Nous référençons des travaux dans ce domaine ainsi que ceux mentionnant le traitement des informations incomplètes. Les relations entre les contraintes de schéma et les contraintes d'intégrité méritent d'être discutées, comme montré dans certaines publications. Nous comparons notre travail avec autres propositions pour la validation de contraintes d'intégrité. Finalement, nous référençons des travaux où les requêtes arbre ou l'idée de projection sur un document XML sont utilisées.

Les contraintes d'intégrité

Dans nos travaux nous avons considéré les clés et les clés étrangères en suivant l'approche des travaux de l'Université de Pennsylvanie (voir, par exemple, [BDF$^+$03]). Ainsi, nos clés sont définies de façon indépendante du type du document.

Dans [LLL02] nous trouvons un des premiers travaux sur les XFD. Les idées proposées sont intéressantes mais les concepts ne sont pas précis. Par exemple, les auteurs n'ont pas défini leur notion d'égalité. Dans [AL02, VLL04, LVL03, HT06, WT05] nous trouvons des propositions plus connues et précises concernant les XFD. Les points de divergence et d'accord entre plusieurs de ces travaux ont été discutés dans [HT06, WT05]. Dans [BHL07, Lim07] notre but est d'introduire une syntaxe homogène pour représenter différents types de contraintes d'intégrité tout en définissant une sémantique capable d'englober plusieurs aspects de ces travaux. Ce mémoire de HDR résume l'approche présentée dans [BCH$^+$07, BHL07] et, en plus, propose l'utilisation des requêtes arbre qui, à mon avis, rend les explications plus simples et ouvre une perspective de recherche intéressante.

Des petites variations existent entre les langages de chemins proposés dans [AL02, VLL04, WT05]. Le langage de chemin dans [AL02, VLL04] spécifie seulement des chemins simples. Le langage de chemins de [WT05] a plus de pouvoir d'expression. Dans [WT05] les chemins peuvent être simples, composés, ascendants ou descendants. Un chemin simple est descendant et ne contient pas de *wildcards* alors qu'un chemin composé consiste en un chemin ascendant suivi d'un chemin descendant. Dans toutes ces approches le langage de chemin utilisé correspond à un langage de requête unaire. L'approche de [HT06] est différente car le langage de chemin est n-aire. En effet, la requête de chemin est représentée par un arbre.

Dans tous les travaux cités auparavant ([AL02, BDF$^+$03, HT06, VLL04, WT05]) les chemins définis dans un langage donné sont ensuite instantiés sur le document XML. Pour cela chaque auteur propose une définition compatible avec ses langages de chemin. Par exemple, la définition de XFD de [AL02] introduit la notion de *tree tuple* qui est une fonction qui associe à un chemin une position d'un arbre XML. À partir de cette notion, les auteurs définissent une arborescence sur l'arbre XML (notion à rapprocher de notre définition 6.1.7) et considèrent des n-uplets (similaire à nos n-uplets de la définition 6.1.8) qui sont comparés pendant la vérification d'une XFD. Dans [VLL04] le concept d'un ensemble de chemins simples et fermé sur préfixe correspond au concept de motif que nous avons utilisé dans [BHL07]. Mais, chez ces auteurs toutes les définitions sont faites sur le concept de chemin simple, descendant. Bref, l'idée (plutôt pédagogique) d'un chemin et d'un parcours arborescent n'est pas utilisée. Comme nous avons déjà dit, la définition de XFD de [HT06] est à rapprocher du concept de requête arbre que nous utilisons ici. La définition des XFD est faite sur les concepts d'homomorphisme, d'isomorphisme et de v-sous-graphe (graphe ou arbre qui possède un nœud de départ v et dont la fin est une feuille). L'isomorphisme est utilisé pour déterminer l'équivalence de deux sous-arbres et cette équivalence est similaire à l'égalité par valeur à partir de la racine (définition 6.1.10). Même si le point de démarrage (un langage de requête n-aire) est différent de celui de [AL02] (un langage de requête unaire), les deux approches sont similaires car les dépendances fonctionnelles sont définies via des structures arborescentes (des parcours arborescents). Une originalité de [WT05] est l'utilisation de différents type d'égalité.

Les problèmes de décision : la satisfaction et l'implication des contraintes

Dans nos travaux, nous n'avons pas considéré les problèmes de décision. Néanmoins, le problème de la satisfaction et de l'implication sont très importants dans plusieurs applications pour assurer la viabilité d'un ensemble de contraintes ou pour minimiser le nombre de contraintes à vérifier lors de l'exécution d'une validation ou encore dans l'analyse de la propagation des contraintes sur des vues. Ces problèmes ont été le sujet de certains travaux que nous commentons ici, tout en essayant de placer nos propositions dans le cadre des résultats publiés.

Problème de la satisfaction (finie) des contraintes : Étant donné un ensemble $\mathcal{C}$ de contraintes dans un

langage L, existe t-il un arbre XML (fini) T qui satisfait $\mathcal{C}$?

Problème de l'implication (finie) des contraintes : Étant donné un ensemble $\mathcal{C}$ de contraintes et une contrainte c, exprimées dans un langage L, est-ce que n'importe quel document (fini) qui satisfait $\mathcal{C}$ satisfait aussi c ? Autrement dit, nous voulons savoir si $\mathcal{C} \models c$?

Dans le cadre du modèle relationnel, les travaux sur les problèmes de la satisfaction et de l'implication sont nombreux et certains résultats méritent d'être rappelés ([AHV95, LL99]). Le problème de la satisfaction pour les clés et plus généralement pour les dépendances fonctionnelles est trivial (considérer une base vide). Mais, en plus, étant donné un ensemble de contraintes contenant des dépendances fonctionnelles standards (la partie gauche n'est pas vide) et des dépendances d'inclusions nous pouvons toujours trouver une instance finie qui satisfait précisément (c'est-à-dire, pas plus) cet ensemble de contraintes (et ses conséquences logiques) [FV83, LL99]. Le problème de l'implication dans les cas de dépendances fonctionnelles (et aussi pour les clés) est décidable en temps linéaire, alors que pour les dépendances d'inclusion le problème est PSPACE complet. Le problème est indécidable si les dépendances fonctionnelles et d'inclusion sont considérées ensemble. Un résultat similaire est obtenu pour l'implication des clés et des clés étrangères. Ces résultats ont motivé des travaux de recherche prenant en compte des sous classes des dépendances d'inclusion comme les dépendances non circulaires ou les dépendances unaires ([LL99]).

Dans le cadre de XML la situation peut être plus compliquée à cause de la structure hiérarchique des données. Comme expliqué dans [BDF$^+$01, BDF$^+$03] certaines propositions de clé ne sont pas satisfaisables de façon finie dû aux restrictions d'existence et d'unicité imposées (ces restrictions n'existent pas pour les autres contraintes). Par exemple, une clé $K = (//, \{k\})$ où le nœud contexte est la racine et le nœud cible est tout autre nœud du l'arbre, impose à tout élément d'avoir une clé k (y compris tout élément ayant le nom k). En d'autres termes, cette contrainte de clé impose une chaîne infinie de k nœuds et ainsi il n'existe pas un document fini capable de satisfaire cette contrainte. La définition d'une *clé forte* de [BDF$^+$03], nous permet de spécifier une clé comme la clé K ci-dessus : (1) le langage de chemin proposé permet la définition des cibles comme dans K et (2) une clé forte impose l'unicité et l'existence des chemins P_i.

Dans notre cas (section 6.2.3), même si nous imposons que chaque chemins P_i existe et soit unique, la spécification d'une telle clé n'est pas possible. En effet, le chemin // ne fait pas partie du langage PL ici considéré[10]. De même, dans nos travaux précédents [BCH$^+$07], alors que le langage de chemin est le même que celui utilisé dans [BDF$^+$01, BDF$^+$03], les chemins P_i des clés sont définis comme spéciaux car ils doivent toujours finir sur un nœud *data* (et donc sans fils). Nos clés sont donc des cas plus restreints des clés fortes de [BDF$^+$01, BDF$^+$03]. À partir des résultats de [BDF$^+$03] nous constatons que notre proposition de clé est satisfiable pour des document finis. Cela concerne aussi le problème d'implication. Dans [BDF$^+$03] nous trouvons la preuve de la décidabilité en PTIME du problème de l'implication (finie) pour les clés.

Dans [FS03] nous trouvons une étude des problèmes de l'implication et de l'implication finie des clés et des clés étrangères (ensemble) sur trois langages de contraintes proposés par les auteurs. Ces langages permettent seulement la définition des clés et des clés étrangères en termes d'attributs XML. Pour un langage permettant l'expression des clés et clés étrangères avec plusieurs attributs (similaires à celles du modèle relationnel), les problèmes de l'implication et de l'implication finie sont indécidables. Néanmoins, ces problèmes sont décidables lorsque nous avons une restriction de clé (c'est-à-dire, pour un type d'élément, il existe un seul ensemble d'attributs X capable de constituer une clé). Pour les autres langages (moins puissants) les problèmes deviennent décidables en temps linéaire.

Une axiomatisation finie pour une classe de XFD est proposée dans [HT06]. Nos dépendances fonctionnelles peuvent exprimer celles proposées par [HT06], mais contrairement à [HT06], notre utilisation des deux types d'égalité nous permet de définir des XFD concernant des nœuds internes. Nous devons donc vérifier si les axiomes de [HT06] s'appliquent dans ce cas ainsi que dans notre traitement des informations incomplètes. Même si [AL02, VLL04, WT05] définissent une forme normale pour XML, ces articles ne proposent pas une axiomatisation pour les XFD. Le problème de l'implication des XFD avec des contraintes de schéma d'une DTD est abordé dans [AL02]. Dans [VSL$^+$04], les auteurs proposent une axiomatisation pour les XID.

[10] La grammaire définissant PL oblige l'existence d'un label l après le *wildcard* //, donc, nous pouvons avoir $\epsilon//l$, mais pas l'inverse

Les informations incomplètes

Les documents XML incomplets, c'est-à-dire ayant des valeurs inconnues, sont mentionnés dans [AL02] et la satisfaction faible ou forte peut être appliquée. Dans [VLL04] la *satisfaction forte* est adoptée. Dans [HT06] les valeurs inconnues sont caractérisées pour être évitées lors de la définition de l'axiome de transitivité. De cette façon, leur axiomatisation s'applique à des XFD complètes (sans *null*). Dans [AL02, HT06, WT05], des questions concernant l'axiomatisation et des formes normal sont considérées.

Les contraintes d'intégrité et de schéma

Un parallèle entre schéma (type du document) et chemin est possible. Un schéma établit la structure voulue d'un document XML. Un arbre XML valide par rapport à un schéma est une instance du schéma. Un chemin P dans un langage de chemin donné peut être vu comme un schéma qui définit un parcours vertical et sans branchement sur un arbre XML. Ainsi, un ensemble de chemins $Path_T$ est un schéma moins contraignant que ceux présentés dans la section 2.2. Réciproquement, un arbre XML est une collection d'instances de chemins qui partent de la racine. L'utilisation d'un ensemble de chemins, plutôt qu'un schéma nous place dans un contexte plus général, car la conformité d'un arbre XML à un ensemble de chemins est moins restrictive que la conformité par rapport à une grammaire LTG ou $STTG$ [VLL04], surtout parce qu'il n'y a pas de notion d'ordre entre chemins.

Pour autant, la conformité d'un arbre par rapport à un ensemble de chemins est une condition suffisante pour la vérification des contraintes d'intégrité. C'est la démarche adoptée dans [BDF$^+$01], qui permet de spécifier des contraintes d'intégrité indépendamment d'un schéma, également dans [AL02], qui utilise l'ensemble de tous les chemins dans une DTD D, noté *Path(D)*. C'est aussi ce que représente le *graphe-schéma* utilisé dans [HT06] et l'ensemble de *chemins légaux* de [VLL04].

Nos travaux prennent en compte cette "indépendance" par rapport aux schémas. Dans [BHL07] nous introduisons la notion de *motif* (qui correspond, en fait, à l'ensemble de chemins simples, fermé par préfixes, défini par une requête `Query`). Pour formaliser la notion de contraintes d'intégrité et leur axiomatisation, dans [BHL07] nous avons considéré que T est conforme à X (noté $T \triangleleft X$), où X peut être *Paths(D)*, un motif, un ensemble de chemins légaux ou un graphe schéma. Autrement dit, étant donnée X, un schéma Υ et un arbre T, l'arbre T peut être non valide par rapport à Υ tout en étant conforme à X (et cela suffit pour valider T en fonction des contraintes d'intégrité). Nous pouvons être intéressés par une validation par rapport aux deux types de contraintes tout en gardant leur indépendance. C'est notre approche dans [BCH$^+$07], où étant donné des contraintes de schéma et des contraintes de intégrité et supposant qu'elles soient cohérentes entre elles, les algorithmes de validation, proposés de façon indépendante, peuvent être intégrés pour permettre la validation de deux types de contraintes dans un seul parcours du document.

La vérification de la cohérence des contraintes n'est pas une question simple, mais nous ne la considérons guère dans nos travaux. En effet, savoir s'il existe un document XML satisfaisant en même temps des contraintes d'intégrité et des contraintes de type données par une DTD est le thème des articles [AFL02, AFL05]. Le but est de valider des spécifications XML statiquement (en temps de compilation). Or, contrairement au modèle relationnel où les clés et les clés étrangères peuvent être spécifiées sans soucis de cohérence avec le schéma, dans le cadre de XML des contraintes de type (une DTD) peuvent interagir avec les clés et les clés étrangères. Cela rend l'analyse de la cohérence des contraintes beaucoup plus compliquée. L'étude présentée dans [AFL02, AFL05] considère plusieurs classes de contraintes et présente les résultats de complexité pour les problèmes de la cohérence des clés, clés étrangères combinées avec les contraintes d'une DTD. Les résultats montrent que pour les classes plus générales le problème est indécidable, restant NP-complet pour la classe de contraintes absolues unaires. Le test de cohérence est linéaire pour la classe contenant seulement les contraintes de clés sur une DTD.

La validation des contraintes d'intégrité

Comme nous avons déjà dit, dans nos travaux, les contraintes d'intégrité sont spécifiées indépendamment des contraintes de type. Nous avons ainsi proposé des algorithmes de validation indépendants. Cette approche est la même que celle suivie par [BDF$^+$01, VLL04, WT05], entre autres. Dans des propositions comme [FKS01] les contraintes de types et les contraintes d'intégrité sont considérées, mais en imposant des restrictions sur la définition des types. La validation incrémentale de contraintes dans [BBG$^+$02] est

fondée sur un générateur de code qui recalcule des formules logiques correspondant à des contraintes. Les informations sur le type permettent l'optimisation. Dans [CDZ02], un validateur pour les contraintes XML est présenté. Même si ce travail est similaire au notre, il se consacre exclusivement aux contraintes de clé. De plus, leur validation incrémentale prend en compte une seule mise à jour et non des mises à jour multiples comme dans nos algorithmes.

Les requêtes arbre et les projections

Dans ce mémoire, des requêtes arbre sont utilisées pour exprimer les contraintes d'intégrité. Cette proposition s'inspire des requêtes arbre proposées dans [Deb05, GI06], mais nous restons dans un cadre beaucoup plus simple : (1) nos requêtes utilisent seulement les axes *descendant* et *père* (par rapport aux axes de XPath) ; (2) une requête arbre est une abréviation d'un ensemble de requêtes de chemins dans un langage comme PL et (3) les nœuds sélectionnés sont seulement ceux nécessaires dans la définition des contraintes. L'idée de requêtes arbre n'est pas nouvelle. Ces requêtes sont, en général, plus sophistiquées que les nôtres car elles utilisent différents axes du langage XPath comme labels possibles pour les arcs.

Parmi les travaux où l'idée de requêtes graphe est présentée, nous pouvons citer [BFK03, Deb05, GI06, GKS04, HT06]. Dans [BFK03] les concepts de *forest pattern* et *tree pattern* sont introduits pour caractériser des requêtes XPath. Une requête graphe sur un document XML coloré permettant l'utilisation des 13 axes est proposée dans [Deb05]. Dans ce travail, la possibilité d'un nœud OR dans la requête graphe est une extension des requêtes conjonctives de [GKS04]. L'évaluation sur un document XML T d'une requête CoreXPath sans négation (cas particulier d'une requête arbre) q a une complexité linéaire ($O(|q|.|T|)$). Ce résultat, publié dans [GKS04], est étendu dans [Deb05] pour les documents colorés. Les requêtes arbre dans [GI06] ont des arcs associés à des expressions régulière sur Σ (l'alphabet de l'arbre XML). Ces requêtes sont descendantes et n-aires. Le but du travail est déterminer statiquement l'impact d'une mise à jour (définie par des requêtes arbre) sur des vues (également définies par des requêtes arbre).

Dans [BCCN06] des règles de déduction sont définies pour pouvoir, statiquement, inférer un *projecteur* à partir d'un chemin XPath P (une requête) et d'une DTD. Le projecteur est un ensemble de labels et correspond en fait à la notion de motif que nous avons utilisé dans [BHL07] et qui peut être obtenu à partir de notre requête `Query`. L'approche [BCCN06] présente un système d'inférence de type pour obtenir le projecteur (ou motif) en prenant en compte les restrictions d'une DTD. Ce formalisme permet aux auteurs de prouver que leur projecteur est correct et minimal (sous certaines conditions imposées sur la DTD). Contrairement à [BCCN06], notre définition de l'ensemble de chemins simples obtenus à partir d'une requête `Query` est non constructive et indépendante du schéma. En effet, chez nous, le motif (ou projecteur) n'est pas calculé explicitement - il est exprimé via les automates d'états finis définis par les contraintes d'intégrité.

Dans [BCCN06], une fois le projecteur calculé, la projection d'un document XML sur ce projecteur peut être calculée par un parcours SAX du document. Le but de [BCCN06] est l'obtention d'une projection capable d'être l'approximation de différentes requêtes. Ainsi, pour répondre à une requête, il n'est pas nécessaire de manipuler tout le document XML mais seulement les parties pouvant influencer la requête. La procédure pour l'obtention de la projection est similaire à notre étape de validation dans laquelle les automates d'états finis correspondant aux contraintes sont utilisés. Plus encore, notre structure auxiliaire (AUX), obtenue lors d'une première validation *from scratch*, peut être vue comme une projection par rapport aux `Query` qui définissent nos contraintes d'intégrité. Contrairement à [BCCN06], nous considérons la mise à jour de cette projection.

6.6 Conclusions

Dans ce chapitre nous avons présenté un formalisme homogène pour la spécification des contraintes d'intégrité exprimable via des chemins linéaires ou des requêtes arbre. La définition de l'instantiation d'une requête arbre (ou la projection d'un document XML sur cette requête) nous permet de définir précisément l'interprétation de chaque type de contrainte, y compris en cas d'information incomplète. Il est intéressant de remarquer qu'une requête arbre définit un motif ([BHL07]), notion similaire au projecteur de [BCCN06].

Nous avons aussi esquissé les principes de la validation incrémentale d'un document par rapport à des contraintes d'intégrité. Ce processus est linéaire sur la taille du document à vérifier, et s'appuie sur

l'utilisation d'automates à états finis et d'une grammaire d'attributs.

Les points communs de notre travail avec, en particulier [BCCN06, GI06], nous permettent d'envisager des perspectives ayant comme but une analyse statique de l'impact des mises à jour (définies par exemple, par des requêtes arbre) sur les contraintes d'intégrité.

Chapitre 7

Premiers pas vers la composition des services web avec PEWS

Dans ce chapitre, complètement indépendant des précédents, je résume nos premiers travaux dans le domaine des services web.

Les services web sont des logiciels autonomes, accessibles via Internet. Ils sont typiquement conçus pour participer à des applications composées où l'interaction avec d'autres services est nécessaire. Pour que l'interaction avec d'autres services soit possible, une description de l'interface du service est nécessaire. L'interaction est donc faite en respectant cette description et en utilisant des messages XML.

Actuellement, WSDL (*Web Services Description Language*) [CCMW01] est le standard pour la description de l'interface d'un service web. Son succès réside dans la séparation claire entre la description abstraite et technique du service. Le langage WSDL permet une description de l'interface visible (ou publiée) du service web. Néanmoins, WSDL considère seulement les aspects statiques d'une interface (c'est-à-dire, la liste des opérations avec les messages d'entrée et de sortie) et non le comportement (l'ordre des opérations). Ainsi, une interface WSDL ne fournit pas toutes les informations nécessaires à la construction de l'interaction avec le service et ne permet pas la vérification de la dynamique du système, contrairement aux besoins exprimés par le groupe de travail W3C [ABPRT04].

Différentes solutions ont déjà été proposées pour ce problème. Ces solutions s'étendent de la proposition de nouveaux langages d'implémentation aux spécifications formelles. Parmi les nouveaux langages permettant la description du comportement d'un service ainsi que la définition de services composés nous pouvons citer WSCI [AAF+02], XLANG [Tha01], WSFL [Ley01], BPML [IB02] et BPEL4WS [ACD+03]. Néanmoins, aucun de ces langages a été introduit avec une sémantique formelle bien définie. En ce qui concerne les spécifications formelles, afin d'aborder les questions intéressantes concernant la composition des services web (telles que la synchronisation, la coordination, la maintenance et la manipulation de la composition), différents formalismes ont été envisagés, comme par exemple, l'algèbre des processus [SBS04], les machines à états finies [BCG+03, HBCS03] ou les réseaux de Petri [HB03].

Dans ce contexte, notre motivation est la proposition d'un langage de spécification simple et formel. Dans nos travaux ([BCHM05, BHM06]), nous introduisons le langage PEWS (*Path Expression for Web Services*), ayant une sémantique formelle bien définie et permettant la spécification du comportement des interfaces des services web simples ou composés, comme complément aux descriptions WSDL. Le langage PEWS donne à l'utilisateur la possibilité de travailler dans les deux niveaux de précision existant en WSDL : un niveau (plus abstrait) qui prend en compte les opérations et un niveau qui considère les messages de chaque opération. PEWS est naturellement adapté au modèle WSDL ; puisque un programme PEWS peut être vu dans deux perspectives : une *dimension contrôle* et une dimension *contrôle plus données*. La dimension contrôle établit seulement l'ordre dans lequel les opérations (ou les services) doivent être exécutés. La dimension *données* ajoute des contraintes sur les restrictions de contrôle, fournissant des opérateurs qui introduisent, implicitement, des restrictions de communication entre les services.

Étant donné la spécification d'une composition en PEWS il est souhaitable de pouvoir vérifier l'existence d'au moins une configuration dans laquelle l'interaction entre les services de la composition fonc-

tionne correctement. Nous sommes ainsi intéressés par la vérification de la correction d'une spécification donnée. Pour cela, nous avons proposé dans [BHM06] l'utilisation de la théorie des traces, introduite dans [Maz87, Maz95]. Les traces montrent de façon séquentielle le comportement non séquentiel d'un système concurrent. Néanmoins, même si les traces nous permettent d'exprimer de façon élégante la communication entre services, l'implémentation des outils de vérification en utilisant cette théorie n'est pas très efficace. Les graphes de dépendances, équivalents aux traces, ont été utilisés pour surmonter cette difficulté [BH08].

Dans ce contexte, notre but à long terme est la construction d'une plate-forme d'aide à la spécification, à l'implémentation et à la manipulation des services. Dans un premier temps, cette plate-forme permettrait à un utilisateur non naïf de tester la spécification d'une composition qu'il élabore à partir des services stockés dans une bibliothèque de services. Ensuite, il serait intéressant d'étudier l'ajout d'autres fonctionnalités permettant la manipulation des compositions, comme par exemple le remplacement d'un service d'une composition par un autre ; avec une vérification incrémentale de la correction de la composition.

Nous envisageons une plate-forme pouvant être utilisée comme un outil de haut et de bas niveau pour la modélisation et l'analyse de services. Comme un outil de bas niveau, elle devrait offrir la possibilité de traduire une spécification PEWS en programme Java (en supposant que les codes des opérations ou des services de base sont disponibles). Comme outil de haut niveau, elle permettrait de faire des tests sur une composition. Cet objectif a été adopté par nos partenaires dans ce travail : le BRGM (Bureau de Recherches Géologiques et Minières, Orléans) qui finance, avec la Région Centre, la bourse de doctorat de Cheikh Ba ; et le groupe de travail de Martin Musicante à l'*Universidade Federal do Rio Grande do Norte*. En effet, les travaux exposés ici sont le résultat de cette collaboration et font partie de la thèse de Cheikh Ba, que j'encadre et dont la soutenance est prévue pour 2008.

Dans ce mémoire, après un rapide aperçu du langage PEWS (section 7.1), nous rappelons les concepts de base concernant la théorie des traces et les graphes de dépendances (section 7.2). Nous montrons comment nous avons étendu les opérations sur les traces aux graphes de dépendances (section 7.3) et, ensuite, nous montrons comment une spécification PEWS est traduite en système de traces ou en ensemble de graphes de dépendances (section 7.4). Quelques propriétés d'une spécification PEWS sont alors discutées (section 7.5). Des travaux liés (section 7.6) et une conclusion (section 7.7) terminent le chapitre.

7.1 Aperçu du langage PEWS

Pour pouvoir placer notre travail dans le domaine des services web, nous adoptons le cadre général proposé dans [SBS04] où les langages liés aux services sont classés en trois catégories, à savoir :

1. La *catégorie des langages abstraits*, utilisés pour le raisonnement. Dans cette catégorie, nous pouvons placer différentes propositions comme l'algèbre de processus, les automates d'états finis, etc.
2. La *catégorie des langages publics ou d'interface* qui contient les langages XML permettant la description du comportement des services.
3. La *catégorie des langages d'implémentation* qui contient les langages utilisés pour l'implémentation des services.

PEWS se place dans la *catégorie des langages abstraits*, alors que sa version XML, appelée XPEWS et présentée dans [BCHM05] fait partie de la *catégorie des langages publics*. Notre proposition permet une traduction automatique d'un programme PEWS en XPEWS.

Différents mécanismes pour la synchronisation de processus ont été proposés comme langage pour la spécification de l'interface de services. Dans ce cadre, PEWS adapte les *predicate path expressions* aux services web. Les *predicate path expressions* [And79] ont été introduites comme un outil pour exprimer la synchronisation des opérations d'un objet. Ces expressions, similaires à des expressions régulières, sont des constructeurs utilisés pour restreindre les suites d'opérations possibles d'un objet. L'utilisation des prédicats dans les *path expressions* permet un contrôle plus fin. Par exemple l'expression $A^{\star}.([P]B + [not\ P]C)$, où A, B et C sont des services et P un prédicat, correspond à un *if-then-else*, c'est-à-dire, l'expression indique que B ou C seront exécutés selon la valeur de vérité de P. L'exécution de B ou C vient après zéro ou plusieurs exécutions de A.

Dans [BCHM05] nous décrivons la syntaxe et la sémantique de PEWS en considérant seulement les opérateurs disponibles dans la dimension contrôle. Dans [MP06], il a été montré que la dimension contrôle de PEWS peut exprimer la plupart des motifs des flux de contrôle définis dans [ATKB03]. L'article [BHM06] introduit la dimension *données plus contrôle* que nous présentons ici.

Pour illustrer notre approche, rappelons d'abord que la sémantique WSDL offre quatre modèles d'interaction, c'est-à-dire, quatre types d'opérations : *one way* (une seule entrée) , *notification* (une seule sortie), *request-response* (entrée-sortie) et *solicit-response* (sortie-entrée).

Considérons maintenant une description WSDL contenant des opérations complémentaires comme `command` et `order` (pour envoyer et recevoir une commande) ; `bill` et `recBill` (pour envoyer et recevoir une facture) ; `payment` et `sendPayment` (pour recevoir et envoyer un paiement) ; `receipt` et `recReceipt` (pour envoyer et recevoir un reçu) et l'opération `sendSellInfo` (pour envoyer le nombre total d'articles vendus). La figure 7.1 montre une partie de cette description WSDL.

```
<message name="messOrder">
  <part name= "prodCode"
        type="xsd:string"/>
  <part name= "quantity"
        type="xsd:integer"/>
</message>

...
```

```
<portType name="WarehousePortType">
     <operation name="order">
          <input message="messOrder"/>
      </operation>

     <operation name="command">
          <output message="messOrder"/>
      </operation>
```

FIG. 7.1: Un extrait d'un fichier WSDL. Description des opérations `order` et `command`.

Nous supposons ensuite les spécifications PEWS des services `OrderTreatment` et `Client`. Dans le cas du service `OrderTreatment` la contrainte à respecter est l'envoie d'un reçu seulement après avoir encaissé le paiement. Dans le cas du service `Client` il est possible d'effectuer un paiement avant de recevoir la facture :

service `OrderTreatment = (order.bill.payment.receipt).sendSellInfo`
service `Client = (command.(recBill.sendPayment + sendPayment. recBill).recReceipt)`

Soit maintenant une composition `S` construite à partir de `OrderTreatment` et `Client` en imposant des contraintes sur les messages (les données) :

service `S = (Client` $<\|>$ `OrderTreatment)`

L'opérateur $<\|>$ indique une exécution en parallèle de deux services. Dans ce cas, les restrictions sur le flux de données ne sont pas strictes : `OrderTreatment` et `Client` peuvent communiquer en synchronisant leurs entrées et sorties mais il n'est pas nécessaire que la synchronisation se fasse pour toutes les entrées et sorties des deux services. Autrement dit, l'opérateur $<\|>$ n'impose pas que la composition soit un système clos (c'est-à-dire, un système où toutes les communications sont internes). La figure 7.2 illustre ces services.

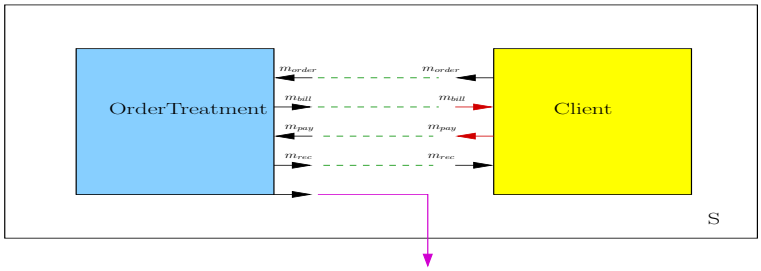


FIG. 7.2: Composition de `OrderTreatment` et `Client`. Le service S a une seule sortie.

Pour pouvoir analyser des compositions en prenant en compte la dimension données, nous avons proposé dans [BHM06] l'utilisation de la théorie des traces, introduite dans [Maz87, Maz95]. Par exemple, dans le cas de notre composition S ci-dessus, le langage de trace obtenu pour le service `OrderTreatment` contient la trace ci-dessous qui montre dans quel ordre les messages sont envoyés ou attendus :

$$[m_{order}?\ m_{bill}!\ m_{pay}?\ m_{rec}!\ m_{SellInfo}!]$$

où ? indique une entrée et ! une sortie. De façon similaire, pour le service `Client`, plusieurs suites de messages sont possibles. La trace :

$$[m_{order}!\ m_{bill}?\ m_{pay}!\ m_{rec}?]$$

où $m_{bill}?$ et $m_{pay}!$ sont considérés comme des messages indépendants, c'est-à-dire, des messages dont l'ordre peut être inversé ; représente, en effet, l'ensemble de suites $\{m_{order}!\ m_{bill}?\ m_{pay}!\ m_{rec}?, m_{order}!\ m_{pay}!\ m_{bill}?\ \ m_{rec}?\}$. En analysant le système de traces de chaque service de la composition S, nous pouvons conclure qu'il existe une configuration permettant un maximum de synchronisations entre les messages de `Client` et `OrderTreatment`. En effet, la synchronisation entre les messages complémentaires est possible et le système résultant contient la trace :

$$[m_{order}\natural\ m_{bill}\natural\ m_{pay}\natural\ m_{rec}\natural\ m_{SellInfo}!]$$

où $\natural$ indique les messages synchronisés, et considérés maintenant comme des messages internes d'un nouveau service S. À partir de l'analyse du système de trace associée à une composition, nous pouvons, par exemple, vérifier si la composition est correcte, c'est-à-dire, s'il existe au moins une possibilité de communication entre les services de la composition qui respecte les contraintes imposées par la spécification.

7.2 La théorie des traces et les graphes de dépendances

Dans cette section, nous rappelons d'abord les principaux concepts dans [Maz87, Maz95] concernant la théorie des traces, en les adaptant à notre contexte. Ensuite nous proposons un rappel des concepts présentés dans [HR95] concernant les graphes de dépendances.

7.2.1 Théorie des traces

La théorie des traces de Mazurkiewicz [Maz87, Maz95] permet la modélisation des événements dans les systèmes concurrents ainsi que l'étude des propriétés concernant l'exécution de ces systèmes. En effet, l'indépendance de certains événements dans un système concurrent permet l'obtention de différentes suites d'événements pour représenter une même exécution. Dans ce cadre, un modèle fondé sur des chaînes de caractères, utilisé pour traiter les systèmes séquentiels, est insuffisant. Les événements concurrents composent un ensemble de séquences représentant une même opération. Le modèle de traces prend en compte ces séquences qui peuvent être représentées par des classes d'équivalence composées de chaînes de caractères.

Exemple 7.2.1 Supposons un service S qui attend comme entrée les informations d'un fournisseur (c'est-à-dire, un message f) et d'un client (c'est-à-dire, un message c), pour pouvoir effectuer un calcul. Les entrées peuvent arriver dans n'importe quel ordre - cela est indiffèrent pour le service. Comment pourvoir représenter la suite des messages en sachant que l'ordre importe peu ? En utilisant des chaînes de caractères, l'ensemble $\{fc, cf\}$ représente les deux suites possibles. La théorie des traces nous permet de faire cette représentation différemment : la trace $[fc]$, où f et c sont des messages indépendants, est équivalente à l'ensemble $\{fc, cf\}$. □

Un *alphabet* A est un ensemble fini de messages. Chaque message m peut être décoré comme $m?$, $m!$ et $m\natural$ pour indiquer, respectivement, une entrée, une sortie et un message interne. Deux messages a et b sont *complémentaires* (c'est-à-dire, $a = \overline{b}$) ssi $a = m!$ et $b = m?$ ou vice-versa. Pour simplifier, nous représentons les décorations seulement dans certains cas absolument nécessaires. Par exemple, l'alphabet $\{m_1?, m_1!, m_1\natural, m_2?, m_2!, m_2\natural\}$ est représenté par $\{m_1, m_2\}$.

Un *alphabet de concurrence* est une paire ordonnée $\Sigma = (A, D)$ où A est un alphabet et D est une relation binaire, symétrique et réflexive sur A, que nous appelons *relation de dépendances* de Σ. En particulier, nous avons l'alphabet de concurrence vide $(\emptyset, \emptyset)$, l'alphabet de concurrence identité (A, ID_A) (avec les plus petites dépendances sur A) et l'alphabet de concurrence complet (A, A^2) (avec les plus grandes dépendances sur A). Étant donnés deux alphabets de concurrence $\Sigma_1 = (A_1, D_1)$ et $\Sigma_2 = (A_2, D_2)$, on définit :

$$\begin{array}{ll} \Sigma_1 \cup \Sigma_2 = & (A_1 \cup A_2, D_1 \cup D_2) \\ \Sigma_1 \cap \Sigma_2 = & (A_1 \cap A_2, D_1 \cap D_2) \\ \Sigma_1 \subseteq \Sigma_2 \Leftrightarrow & (A_1 \subseteq A_2, D_1 \subseteq D_2) \end{array}$$

La relation $I_\Sigma = A^2 \setminus D$ est connue par le nom de *relation d'indépendance* de Σ. Bien entendu, I_Σ est une relation symétrique et non réflexive. Deux symboles a, b dans A sont mutuellement dépendants dans Σ si $(a, b) \in D$; sinon ils sont indépendants.

Exemple 7.2.2 [Maz87, Maz95] : Let $\Sigma = (\{a, b, c\},\ \{a, b\}^2 \cup \{a, c\}^2)$ un alphabet de concurrence. La relation $I_\Sigma = \{(b, c),\ (c, b)\}$ indique que seulement b, c sont indépendants. □

Étant donné un alphabet de concurrence Σ, on définit $\equiv_\Sigma$ une relation d'équivalence définie sur le monoïde[1] $(A^*, \cdot, \epsilon)$. La relation d'équivalence $\equiv_\Sigma$ est définie de façon que, étant donné $a, b \in A$, nous avons : $(a, b) \in I_\Sigma \Rightarrow ab \equiv_\Sigma ba$. En d'autres termes, si deux symboles de l'alphabet sont indépendants, alors ils peuvent être permutés car l'ordre selon lequel ils apparaissent dans une suite peut être modifié sans changer la sémantique. Ainsi, $w' \equiv_\Sigma w''$ ssi il existe une suite finie de mots $(w' = w_0, w_1, \ldots, w_n = w'')$ (avec $n \geq 0$) telle que pour chaque $1 \leq i \leq n$, il existe des mots u, v et des symboles a, b tels que $(a, b) \in I_\Sigma$ avec $w_{i-1} = uabv$ et $w_i = ubav$.

À partir de la définition de $\equiv_\Sigma$ nous pouvons bien comprendre qu'une *trace* correspond à une classe d'équivalence contenant toutes les chaînes de A^* qui sont équivalentes. La trace engendrée par la chaîne w est dénoté $[w]$ (où $[w]_\Sigma$ si nous avons besoin d'expliciter l'alphabet de concurrence). Par exemple, si $(a, b) \in I_\Sigma$, la trace $[ab] = \{ab, ba\}$ et la trace $[aab] = \{aab, aba, baa\}$. Nous rappelons les propriétés suivantes de la théorie des traces :

- L'alphabet de la chaîne w_1 est aussi l'alphabet de la trace $[w_1]$ (appelé $\alpha_{[w_1]}$). Dans le contexte des services web, nous ferons aussi référence aux alphabets d'entrée $\alpha^{in}_{[w_1]}$, de sortie $\alpha^{out}_{[w_1]}$ et interne $\alpha^{h}_{[w_1]}$ contenant les symboles de trace décorés par ?, ! et ♮, respectivement.
- $[u].[w] = [uw]$, où u et w sont des chaînes de symboles d'un alphabet donné.
- $t.[\epsilon] = [\epsilon].t = t$, où t est une trace.
- $t_1.(t_2.t_3) = (t_1.t_2).t_3$, pour les traces t_i (avec $i = 1, 2, 3$).
- Pour n'importe quelles traces β_1, β_2, γ_1 et γ_2 nous avons $\beta_1\beta_2 = \gamma_1\gamma_2$ ssi il existe des traces t_1, t_2, t_3, t_4 telle que $\beta_1 = t_1.t_2$, $\beta_2 = t_3.t_4$, $\gamma_1 = t_1.t_3$, $\gamma_2 = t_2.t_4$ et $\alpha_{t_2} \times \alpha_{t_3} \subseteq I_\Sigma$.

Un préfixe d'une trace t dans Σ est une trace t_0 pour laquelle il existe une trace t_1 tel que $t = t_0.t_1$. Nous notons $\mathit{Prefix}(t)$ l'ensemble de tous les préfixes de t. Nous pouvons établir une relation d'ordre partiel (une relation préfixe) $\preceq$ telle que $t_1 \preceq t_2 \Leftrightarrow t_1 \in \mathit{Prefix}(t_2)$. Nous pouvons ainsi établir un ordonnancement qui commence avec le préfixe $[\epsilon]$ (la trace vide) et continue en augmentant la chaîne de symboles. Il est important de remarquer qu'il s'agit d'un ordonnancement partiel, puisque chaque préfixe de traces est aussi une trace et représente ainsi toute une classe d'équivalence. Ainsi, un préfixe d'une trace t peut être vu comme une exécution possible qui démarre à l'état initial $[\epsilon]$ et qui continue, en construisant la chaîne selon l'ordre des préfixes de la trace jusqu'à l'obtention de l'état d'exécution représenté par la trace t.

Nous dénotons par $\Pi_{\Sigma_2}[t]$ la (trace) projection de la trace $[t]_{\Sigma_1}$ sur l'alphabet de concurrence Σ_2. Le résultat de cette projection est une trace sur l'alphabet de concurrence $\Sigma = \Sigma_1 \cap \Sigma_2 = (A, D)$ définie comme $\Pi_{\Sigma_2}([t]_{\Sigma_1}) = [\Pi_A(t)]_\Sigma$. Nous rappelons que $\Pi_A(t)$ est une chaîne où tous les symboles qui ne sont pas dans A sont supprimés.

Pour chaque langage $L \subseteq A^*$, nous définissons le langage de trace $[L]_\Sigma = \{[w]_\Sigma \mid w \in L\}$. Nous dénotons $\Theta(\Sigma)$ l'ensemble de toutes les traces sur Σ. Un langage de traces T est fermé par préfixe si $T = \mathit{Prefix}(T)$. Nous rappelons les propriétés suivantes, où X et Y sont des langages de mots :

- $[\emptyset] = \emptyset$
- $[X].[Y] = [X.Y]$
- $X \subseteq Y \Rightarrow [X] \subseteq [Y]$
- $[X] \cup [Y] = [X \cup Y]$
- $[X]^* = [X^*]$

Un système de traces $\mathcal{T}$ est une paire ordonnée (Σ, T) où Σ est un alphabet de concurrence et T est un langage de traces dans $\Theta(\Sigma)$.

[1] Comme d'habitude A^* est l'ensemble de toutes les chaînes d'éléments de A, . est l'opération de concaténation et l'élément neutre ϵ est la chaîne vide.

7.2.2 Graphes de dépendances

Au lieu de voir une trace comme un ensemble d'objets (mots), nous pouvons la voir comme un unique objet : le *graphe de dépendances*. Les graphes de dépendances sont des représentations graphiques des chaînes de symboles qui rendent explicite l'ordre de l'occurrence de ces symboles dans les traces. Étant donné un alphabet de concurrence $\Sigma = (A, D)$, chaque mot x sur A correspond à un graphe de dépendances (appelé graphe de dépendances *concret*). Il s'agit d'un graphe fini, dirigé, acyclique dont les nœuds sont étiquetés par des labels (ou symboles) dans A. Dans ce graphe, deux nœuds sont liés par un arc ssi ils sont des nœuds distincts et étiquetés par des symboles dépendants. La définition ci-après formalise cette notion.

Définition 7.2.1 - Graphe de dépendances ([HR95]) : Soient $\Sigma = (A, D)$ un alphabet de concurrence et $x = a_1 \dots a_n$ un mot sur A, avec $n \geq 0$. Le graphe de dépendances de x, dénoté $\langle x \rangle_\Sigma$, est le graphe acyclique, orienté et étiqueté $G = (V, E, l)$, où : (1) $V = \{1, \dots, n\}$; (2) pour tout $i, j \in V$, $(i, j) \in E$ ssi $i < j$ et $(a_i, a_j) \in D$, et (3) pour tout $i \in V$, $l(i) = a_i$. Le graphe vide, c'est-à-dire, le graphe où l'ensemble de nœuds est vide est noté $G_\emptyset$. Le graphe $G_\epsilon = (\{id\}, \emptyset, \{(id, \epsilon)\})$, où id est un identificateur de nœud spécial, correspond à la trace $[\epsilon]$. □

Comme dans la définition 7.2.1 la relation D est réflexive, les nœuds distincts du graphe de dépendances ayant un même label (sans prendre en compte les décorations) sont connectés.

Si nous faisons abstraction des identités des nœuds dans un graphe de dépendances concret (définition 7.2.1) nous obtenons les *graphes abstraits*, c'est-à-dire, des classes de graphes isomorphes. En d'autres termes, un graphe de dépendances abstrait représente toutes les chaînes qui appartiennent à la trace. *Un graphe g est un graphe de dépendances sur Σ pour x, s'il est isomorphe à $\langle x \rangle_\Sigma$* (nous notons $g \cong \langle x \rangle_\Sigma$).

Ainsi, les graphes de dépendances ne sont pas vraiment des représentations des mots, mais des représentations de traces. En effet, deux mots sont des éléments d'une même trace ssi leurs graphes de dépendances sont isomorphes [HR95]. Étant donné un graphe de dépendances $\langle x \rangle_\Sigma$, nous pouvons trouver de façon non déterministe tous les mots w_k appartenant à la trace $[x]$ en effectuant un tri topologique [HR95] : consécutivement, pour $i = 1, \dots, n$ le i^{eme} élément du mot w_k est un des nœuds minimaux du graphe résultant de $\langle x \rangle_\Sigma$ après avoir supprimer les éléments $1^{er}, \dots, (i-1)^{eme}$ déjà choisis (avec leurs arcs entrants). Un *nœud minimal* d'un graphe est un nœud sans arcs entrants. Les mots qui peuvent être lus à partir du graphe abstrait composent le langage du graphe. Remarquer qu'à partir d'un graphe nous pouvons aussi obtenir tous les préfixes du langage.

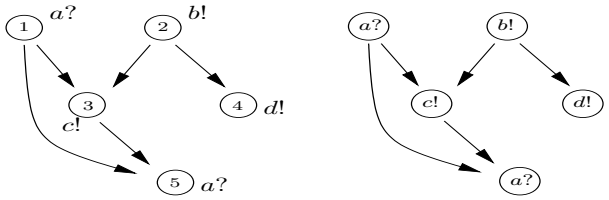


FIG. 7.3: Graphe de dépendances $\langle x_1 \rangle_{\Sigma_1}$ et les différentes manières de le voir : graphe concret et graphe abstrait

Exemple 7.2.3 Soit $\Sigma_1 = (A_1, D_1)$ un alphabet de concurrence tel que $A_1 = \{a, b, c, d\}$ et $D_1 = \{(a, c)^2, (b, c)^2, (b, d)^2\}$. La figure 7.3 montre le graphe concret pour le mot $x_1 = a?.b!.c!.d!.a?$ ainsi que sa version abstraite. Ses nœuds minimaux sont 1 et 2, c'est-à-dire, les nœuds sans arcs entrants. Remarquer que les nœuds minimaux représentent des messages indépendants. Le graphe abstrait représente la trace $t_1 = [a?.b!.c!.d!.a?]$ laquelle contient non seulement le mot x_1 mais aussi $x_2 = b!.a?.d!.c!.a?$, $x_3 = a?.b!.d!.c!.a?$, etc. □

Théorème 7.2.1 *[HR95] : Soit $\Sigma = (A, D)$ un alphabet de concurrence et soit x et y des mots sur A. Alors $\langle x \rangle_\Sigma \cong \langle y \rangle_\Sigma$ ssi $x \equiv_\Sigma y$.* □

Dans ce contexte, un système de traces peut être défini comme un ensemble de graphes de dépendances sur Σ. Soit $\mathcal{T} = (\Sigma, T)$ un système de traces où le langage de traces est $T = \{[x_1], \dots, [x_n]\}$. Nous définissons $\mathcal{G} = \{G_1, \dots, G_n\}$ où chaque G_i est le graphe de dépendances $\langle x_i \rangle_\Sigma$. Nous disons que $\mathcal{G}$ est un graphe de dépendances pour un système de trace (TSDG, pour simplifier).

7.3 Les opérations de base

7.3.1 Du coté des traces

Dans cette section nous définissons des opérations spéciales sur les systèmes de traces. La première d'entre elles est le produit *shuffle*. Intuitivement, le *shuffle* mélange les traces des systèmes pour composer un système de traces.

Définition 7.3.1 : Shuffle des systèmes de traces - Soient $\Sigma_1 = (A_1, D_1)$ et $\Sigma_2 = (A_2, D_2)$ deux alphabets concurrents. Étant donné des systèmes de traces $\mathcal{T}_1 = (\Sigma_1, T_1)$ et $\mathcal{T}_2 = (\Sigma_2, T_2)$ le *shuffle* $\mathcal{T}_1 \otimes \mathcal{T}_2$ est défini comme le système de traces $\mathcal{T} = (\Sigma, T)$ tel que : $\Sigma = \Sigma_1 \cup \Sigma_2$ et

$$T = \{[\gamma_1.\beta_1.\gamma_2.\beta_2 \ldots \gamma_n.\beta_n] \mid \gamma_i \in A_1^*,\ \beta_i \in A_2^*, \text{ and } [\gamma_1.\gamma_2 \ldots \gamma_n] \in T_1 \text{ and } [\beta_1.\beta_2 \ldots \beta_n] \in T_2\}. \quad \square$$

Exemple 7.3.1 Soient $\mathcal{T}_1 = ((\{m_1, m_2\}, ID), (\{[m_1!.m_2?]\})$ et $\mathcal{T}_2 = ((\{m_1, m_2\}, ID), \{[m_1?m_2!]\})$. Le *shuffle*, $\mathcal{T}_1 \otimes \mathcal{T}_2 = ((\{m_1, m_2\}, ID), \{[m_1!m_2?m_1?m_2!]\})$. $\square$

La synchronisation est différente du *shuffle* car les messages complémentaires sont appariés (*matched*) alors que les messages non complémentaires sont mélangés, comme dans un *shuffle*.

Dans la définition ci-dessous, $\Sigma|_\beta$ est un alphabet de concurrence Σ restreint aux symboles trouvés dans le mot β.

Définition 7.3.2 : Synchronisation des systèmes de traces - Soient $\mathcal{T}_1 = (\Sigma_1, T_1)$ et $\mathcal{T}_2 = (\Sigma_2, T_2)$ deux systèmes de traces. Pour chaque $t_1 \in T_1$ et $t_2 \in T_2$ tel que $t_1 = [\gamma_1.a_1 \ldots \gamma_n.a_n.\gamma_{n+1}]$ et $t_2 = [\beta_1.b_1 \ldots \beta_n.b_n.\beta_{n+1}]$, où a_i et b_i sont des messages complémentaires et γ_i et β_i sont des chaînes n'ayant pas deux messages complémentaires (c'est-à-dire, γ_i et β_i ne contiennent pas $m?$ et $m!$), nous notons $\mathcal{T}_{\gamma_i,\beta_i}$ le système de traces obtenu par le *shuffle* $\mathcal{T}_{\gamma_i} \otimes \mathcal{T}_{\beta_i}$ où $\mathcal{T}_{\gamma_i} = (\Sigma_1|_{\gamma_i}, \{[\gamma_i]\})$ et $\mathcal{T}_{\beta_i} = (\Sigma_2|_{\beta_i}, \{[\beta_i]\}))$. La synchronisation $\mathcal{T}_1 \circledS \mathcal{T}_2$ est définie par le système de traces $\mathcal{T} = (\Sigma, T)$ tel que $\Sigma = \Sigma_1 \cup \Sigma_2$ et

$$T = \{[t] \mid [t] = [\nu_1.c_1\natural \ldots \nu_n.c_n\natural.\nu_{n+1}] \text{ où le nombre de } c_i\natural \text{ est maximal et}$$
$$\nu_i \text{ appartient au langage de traces de } \mathcal{T}_{\gamma_i,\beta_i}\} \quad \square$$

La définition 7.3.2 est un peu différente de celle présentée dans [Maz87, Maz95] car elle prend en compte les décorations des messages (entrée et sortie) pour les transformer en messages internes. Remarquer aussi que, par soucis d'efficacité, nous sommes intéressés seulement par des résultats permettant de synchroniser le plus grand nombre de messages.

Exemple 7.3.2 Soient $\mathcal{T}_1 = ((\{m_1, m_2\}, ID), \{[m_1!m_2?]\})$ et $\mathcal{T}_2 = ((\{m_1, m_2\}, ID), \{[m_2!m_1?]\})$. La synchronisation $\mathcal{T}_1 \circledS \mathcal{T}_2 = ((\{m_1, m_2\}, ID), \{[m_1\natural m_2\natural]\})$.

Nous considérons maintenant un alphabet de concurrence complet. Soient $\mathcal{T}_1 = ((\{m_1, m_2\}, \{m_1, m_2\}^2)$, $\{[m_1!m_2?]\})$ et $\mathcal{T}_2 = ((\{m_1, m_2\}, \{m_1, m_2\}^2)$, $\{[m_2!m_1?]\})$. La synchronisation $\mathcal{T}_1 \circledS \mathcal{T}_2$ donne comme résultat $((\{m_1, m_2\}, \{m_1, m_2\}^2)$, $\{[m_1!m_2\natural m_1?], [m_2!m_1\natural m_2?]\})$. $\square$

Il est important de remarquer que si un des systèmes de traces est vide, alors les résultats du *shuffle* ou de la synchronisation coïncide avec le système non vide.

Dans certaines situations, nous souhaitons imposer la synchronisation sur tous les messages d'un ensemble préétabli. Pour exprimer ce type de restriction, nous introduisons un deuxième opérateur de synchronisation, que nous appelons synchronisation par rapport à un alphabet. Cet opérateur diffère de l'opération $\circledS$ car il impose la synchronisation de tous les éléments d'un alphabet A.

Définition 7.3.3 : Synchronisation par rapport à un alphabet - Soit l'alphabet A l'ensemble fini des messages $\{c_1, \ldots, c_n\}$. Dans la définition 7.3.2, considérons que dans les traces t_1 et t_2, les messages a_i et b_i soient complémentaires et concernent *tous* les messages $c_i \in A$. De plus γ_i et β_i sont des chaînes de messages qui ne contiennent *aucun message* de A.

La synchronisation $\mathcal{T}_1 \circledS_A \mathcal{T}_2$ est le système de trace $\mathcal{T} = (\Sigma, T)$ tel que $\Sigma = \Sigma_1 \cup \Sigma_2$ et

$T = \{[t] \mid [t] = [\nu_1.c_1\natural \ldots \nu_n.c_n\natural.\nu_{n+1}]$où $c_1, \ldots, c_n \in A$ est le résultat de l'appariement (*matching*) des messages complémentaires dans t_1, t_2 ; et $[\nu_i] \in \mathcal{T}_{\gamma_i,\beta_i}\}$ $\square$

Exemple 7.3.3 Soient $\mathcal{T}_1 = ((\{m_1, m_2\}, ID), \{[m_1!.m_2?]\})$ et $\mathcal{T}_2 = ((\{m_1, m_2\}, ID), \{[m_1?.m_2!]\})$. Soit $\alpha = \{m_1, m_2\}$. Dans ce cas, la synchronisation $\mathcal{T}_1 \circledS_\alpha \mathcal{T}_2 = \mathcal{T}_1 \circledS \mathcal{T}_2$ comme dans l'exemple 7.3.2.

Nous considérons maintenant un alphabet de concurrence complet. Soient $\mathcal{T}_1 = (\{\{m_1, m_2\}, \{m_1, m_2\}^2)$, $\{[m_1!m_2?]\})$ et $\mathcal{T}_2 = (\{\{m_1, m_2\}, \{m_1, m_2\}^2)$, $\{[m_2!m_1?]\})$. La synchronisation $\mathcal{T}_1 \circledS_\alpha \mathcal{T}_2$ donne comme résultat $(\{\{m_1, m_2\}, \{m_1, m_2\}^2), \emptyset)$, puisqu'il n'est pas possible d'obtenir une trace où tous les messages de α ont été synchronisés. □

Remarques :
• Par abus de notation nous allons utiliser aussi les opérateurs $\circledS_\alpha$ et $\circledS$ pour indiquer la synchronisation de deux traces (au lieu des systèmes de traces). Par exemple, $[m_1!.m_2?] \circledS [m_1?.m_2!] = [m_1\natural.m_2\natural]$.
• Dans la suite de ce chapitre, nous verrons que certaines opérations de notre approche sont appliquées seulement à des alphabets disjoints. Par exemple, nous considérons la synchronisation pour des systèmes de traces dont les alphabets coïncident seulement sur les symboles concernés par la synchronisation.

Itération d'un langage de traces : Étant donné un langage de trace T, nous rappelons le concept d'itération des traces de [Maz87, Maz95], noté $T^\star$. Dans $T^\star$ l'itération est faite de façon indépendante, pour chaque composant de la connexion. Selon [HR95], l'itération d'un langage de trace est obtenue de la façon suivante : (i) la décomposition des éléments du langage en composants connectés (notion qui est plus claire en considérant les graphes) et (ii) l'utilisation de la fermeture de Kleene $(*)$ sur le langage decomposé. Par exemple, soit $\mathcal{T} = (\{\{a, b\}, ID), \{[ab]\})$. Nous avons $\mathcal{T}^\star = (\{\{a, b\}, ID), \{[\epsilon], [ab], [abab], [baabba]$, $[aaaabbbb], \ldots\})$.

7.3.2 Du coté des graphes de dépendances

Pour décrire la composition des services web il est souvent nécessaire de spécifier que l'exécution de deux services doit suivre un ordre établi ou qu'une synchronisation des services est nécessaire. Dans la suite nous définissons donc la connexion et la synchronisation sur des graphes de dépendances. Dans les deux cas nous considérons des graphes de dépendance disjoints par rapport à un alphabet. Dans les définitions ci-après, il faut rappeler qu'un graphe de dépendance G est construit par rapport à un alphabet de concurrence $\Sigma = (A, D)$.

Définition 7.3.4 - *Graphes de dépendances disjoints :* *Soient deux graphes de dépendances $G_1 = (V_1, E_1, l_1)$ et $G_2 = (V_2, E_2, l_2)$ où $V_1 \cap V_2 = \emptyset$. Les graphes G_1 et G_2 sont disjoints si pour tout $v_1 \in V_1$ et $v_2 \in V_2$, $l(v_1) \neq l(v_2)$ (mise à part la décoration). En d'autres termes G_1 et G_2 sont disjoints si $A_1 \cap A_2 = \emptyset$* □

Définition 7.3.5 - *Connexion des graphes de dépendances disjoints :* *Soient $G_1 = (V_1, E_1, l_1)$ et $G_2 = (V_2, E_2, l_2)$ deux graphes de dépendances disjoints (différents de $G_\emptyset$ et de G_ϵ) tels que $v_1 < v_2$ pour tout $v_1 \in V_1$ et $v_2 \in V_2$. Une connexion de G_1 et G_2, exprimée par $G_1.G_2$, est un graphe de dépendances $G = (V, E, l)$ où : $V = V_1 \cup V_2$; $E = E_1 \cup E_2 \cup E'$ avec $E' = \{(v_1, v_2) \mid v_1 \in V_1 \wedge v_2 \in V_2\}$ et* $l(v) = \begin{cases} l_1(v) & \text{if } v \in V_1 \\ l_2(v) & \text{if } v \in V_2. \end{cases}$
De plus, nous définissons $G_1.G_\epsilon = G_\epsilon.G_1 = G_1$ et $G_1.G_\emptyset = G_\emptyset.G_1 = G_\emptyset$ □

La connexion de G_1 et G_2 peut être faite en temps $O(|V_1| \times |V_2|)$.

Remarque : **La concaténation comme base pour l'itération**
La définition 7.3.5 peut être adaptée pour introduire la *concaténation* $\odot$, définie dans [Maz87] et qui sera la base de l'opération d'itération sur les graphes. Soit un alphabet de concurrence $\Sigma = (A, D)$ et deux graphes de dépendances $G_1 = (V_1, E_1, l_1)$ et $G_2 = (V_2, E_2, l_2)$ (pas forcement disjoints) sur Σ, tels que $V_1 \cap V_2 = \emptyset$ et pour tout $v_1 \in V_1$ et $v_2 \in V_2$, $v_1 < v_2$. La concaténation $G = G_1 \odot G_2$ est un graphe similaire au graphe G de la définition 7.3.5, sauf pour les arcs, car nous avons $E = E_1 \cup E_2 \cup \{(v_1, v_2) \in V_1 \times V_2 \mid (l_1(v_1), l_2(v_2))$ (sans prendre en compte la décoration) est une dépendance de $D\}$.

Ci-dessous nous définissons la synchronisation par rapport à un alphabet α donné. Cette synchronisation est applicable dans un contexte plus restreint que les définitions générales de la section précédente (définitions 7.3.2 et 7.3.3). Ici, nous partons du principe que seulement les noeuds dont le label correspond à un symbole dans α sont à synchroniser et tous les autres sont des symboles distincts. Par exemple, soit $\alpha = \{a\}$ et soient G_1 et G_2 des graphes dont les nœuds ont des labels $a!, a!, b!$ et $a?, a?, b!$, respectivement. La synchronisation de G_1 et G_2 par rapport à α n'est pas définie.

Définition 7.3.6 - Synchronisation des graphes de dépendances par rapport à un alphabet : Soit α un alphabet. Soient $G_1 = (V_1, E_1, l_1)$ et $G_2 = (V_2, E_2, l_2)$ deux graphes de dépendances disjoints sur $A = ((A_1 \cup A_2) \setminus \alpha)$. Une synchronisation $G_1 \,Ⓢ_\alpha\, G_2$ est un graphe de dépendances $G = (V, E, l)$ tel qu'il existe une fonction $f : V_1 \cup V_2 \to V$ qui respecte toutes les propriétés ci-dessous où $G_i = (V_i, E_i, l_i)$ pour $i = 1, 2$:

- Pour tout $v \in V_i$ tel que $l_i(v) \notin \alpha$, il existe un nœud $f(v) \in V$ pour lequel $l(f(v)) = l_i(v)$.
- Pour tout $v \in V_i$ tel que $l_i(v) \in \alpha$ (où $i \in \{1, 2\}$), il existe un nœud $u \in V_j$ (où si $i = 1$ alors $j = 2$, sinon $j = 1$) tel que $l_i(v) = \overline{l_j(u)}$ et il existe un seul nœud $f(v) \in V$ tel que $f(v) = f(u)$ avec $l(f(v)) = m\natural$ (où m est le message qui doit synchroniser).
- Pour tous les arcs $(v, u) \in E_i$ il existe un arc du nœud $f(v)$ au nœud $f(u)$ dans le graphe G.
- La fonction f est surjective et :
  - Chaque $f(v) \in V$ tel que $l(f(v)) \notin \alpha$, correspond à un nœud v dans V_1 ou dans V_2, mais non dans les deux, ayant le même label (c'est-à-dire, $l(f(v)) = l_i(v)$)).
  - Pour tout $f(v) \in V$ tel que $l(f(v)) \in \alpha$ et $l(f(v)) = m\natural$, il existe un unique nœud $v_1 \in V_1$ tel que $f(v) = f(v_1)$ et $l_1(v_1) = m?$ (ou $l_1(v_1) = m!$) et un unique nœud $v_2 \in V_2$ tel que $f(v) = f(v_2)$ et $l_2(v_2) = \overline{l_1(v_1)}$.
  - Pour tout arc $(f(v), f(u)) \in E$ il existe un arc de v à u dans V_1 ou dans V_2. □

L'algorithme pour calculer la synchronisation parcourt en parallèle les deux graphes via le tri topologique, en testant les nœuds minimaux et en construisant un nouveau graphe de dépendances qui correspond à la synchronisation. Cet algorithme permet de calculer la synchronisation de deux graphes en temps $O(|G_1| + |G_2|)$ où $|G_i| = |V_i| + |E_i|$ indique la taille d'un graphe G_i.

Exemple 7.3.4 Soit $\alpha = \{a, b\}$. La figure 7.4 montre les graphes G_1, G_2 et leur synchronisation $G = G_1 \,Ⓢ_\alpha\, G_2$. □

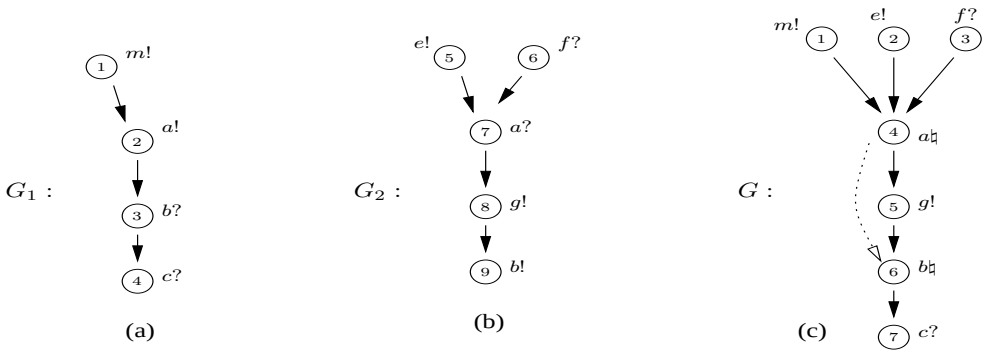


FIG. 7.4: Synchronisation de G_1 et G_2 par rapport à $\alpha = \{a, b\}$.

Exemple 7.3.5 Soit $\alpha = \{a, b\}$. La figure 7.5 montre les graphes G_1, G_2. Dans ce cas la synchronisation n'est pas possible. En effet, si nous considérons la synchronisation $G_1 \,Ⓢ_\alpha\, G_2$; nous pourrions imaginer que le nœud 1 de G_1, ayant comme complémentaire le nœud 4 de G_2 soit associé au nœud 1 de G. Dans ce cas, $f(1) = 1$. Ensuite, nous pourrions proposer que le nœud 1 de G_1, ayant comme complémentaire le nœud 5 de G_2 soit toujours associé au nœud 2 de G. Mais, dans ce cas, nous devons avoir $f(1) = 2$. Cela n'est pas une synchronisation car f (de la définition 7.3.6) n'est pas une fonction.

Si nous considérons la synchronisation $G_2 \,Ⓢ_\alpha\, G_1$, nous pourrions imaginer que les nœud 4 et 5 de G_2, ayant comme complémentaire le nœud 1 de G_1 soient tous les deux, associés au nœud 1 de G (figure 7.5-d). Autrement dit, $f(4) = f(5) = f(1) = 1$. Dans ce cas, G n'est pas un graphe de dépendances car nous avons un un cycle (figure 7.5-d). □

Le produit *shuffle* peut être définit de façon similaire à la synchronisation sur un alphabet (définition 7.3.6). Pour cela il faudrait considérer une fonction f bijective : chaque nœud de $V_1 \cup V_2$ est associé à un nœud dans $V (= V_1 \cup V_2)$ et vice-versa. La définition de la synchronisation Ⓢ est légèrement différente de la définition de $Ⓢ_\alpha$ (définition 7.3.6) : au lieu d'utiliser un alphabet α nous utilisons l'ensemble maximal de synchronisation introduit dans la définition suivante. Dans les deux synchronisations Ⓢ et $Ⓢ_\alpha$; si β ou α sont vides, le résultat est un graphe de dépendances représentant un *shuffle*.

Définition 7.3.7 - Ensemble maximal de synchronisation : Soient $G_1 = (V_1, E_1, l_1)$ et $G_2 =$

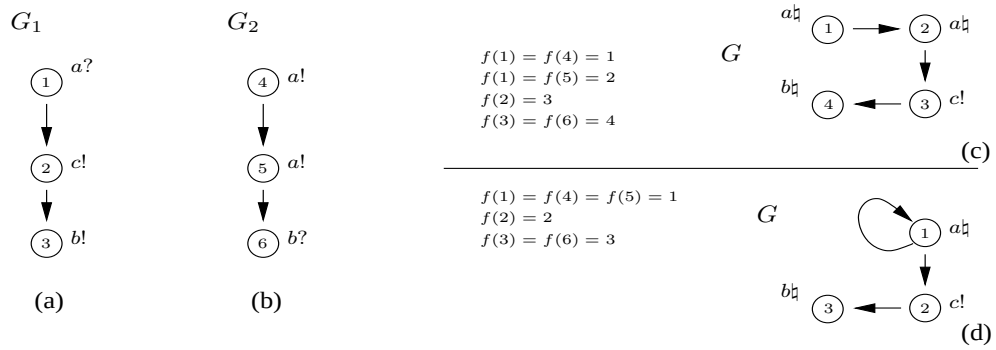


FIG. 7.5: Synchronisation impossible de G_1 et G_2 par rapport à $\alpha = \{a, b\}$.

(V_2, E_2, l_2) deux graphes de dépendances tels que $V_1 \cap V_2 = \emptyset$. Un ensemble maximal de synchronisation est un ensemble $\beta = \{(v_1^1, v_2^1), (v_1^2, v_2^2), \ldots, (v_1^n, v_2^n)\}$ où toutes les propriétés suivantes sont respectées :

1. Chaque couple dans β est composé des nœuds $v_1^i \in V_1$ et $v_2^i \in V_2$ $(i, j \in [1, n])$ tel que : (a) $l_1(v_1^i) = \overline{l_2(v_2^i)}$ et (b) pour tout $i \neq j$ nous avons $v_1^i \neq v_1^j$ et $v_2^i \neq v_2^j$.
2. Pour tous couples (v_1^i, v_2^i) et (v_1^j, v_2^j) dans β tels que $i, j \in [1, n]$ et $i \neq j$, si $v_1^i < v_1^j$ alors $v_2^i < v_2^j$ ou G_2 n'a aucun chemin reliant v_2^i et v_2^j.
3. Il n'existe aucun autre ensemble β' maximal de synchronisation tel que $\beta \subset \beta'$. □

Ainsi, pour obtenir $G = G_1 \circledS G_2$ deux étapes doivent être effectuées : (1) Trouver les ensembles maximaux de synchronisation β selon la définition 7.3.7. Ces ensembles indiquent où la synchronisation doit être faite. Remarquer que β fournit les identificateurs des nœuds dans chaque graphe G_1 et G_2. (2) Effectuer la synchronisation de G_1 et G_2 sur chaque ensemble β. Pour atteindre ce but, nous pouvons utiliser un algorithme similaire à celui présenté dans [BH08] : le raisonnement est le même mais la procédure doit prendre en compte l'identité des nœuds et non les labels.

Exemple 7.3.6 La figure 7.6 montre les graphes G_1, G_2' et leur synchronisation $G' = G_1 \circledS G_2'$. Par la définition 7.3.7, nous avons $\beta_A = \{(2, 7), (3, 9), (4, 11)\}$ et $\beta_B = \{(2, 7), (3, 9), (4, 10)\}$. Remarquer que l'ensemble $\{(2, 7), (3, 9), (4, 5)\}$ n'est pas un ensemble maximal de synchronisation car il ne respecte pas la seconde condition de la définition 7.3.7. La figure 7.6 illustre la synchronisation sur β_A. L'algorithme parcourt les deux graphes, dans l'ordre topologique. □

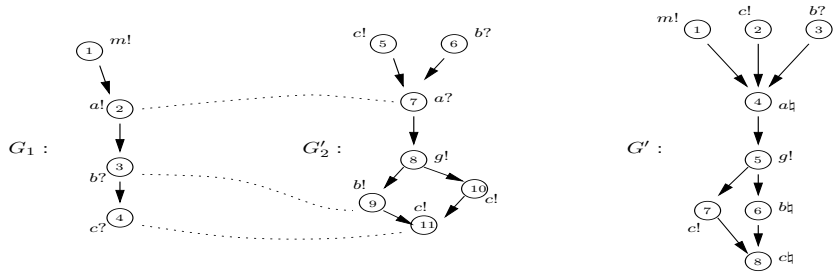


FIG. 7.6: Synchronisation de G_1 et G_2' par rapport à β_A

7.4 Composition des services avec PEWS

Comme nous l'avons déjà dit, PEWS permet la spécification des services composés à partir des services de base (spécifiés aussi avec PEWS). La spécification d'une composition est faite via les différents opérateurs de PEWS et peut être présentée selon deux dimensions différentes. La *dimension de contrôle* considère des opérateurs qui imposent des contraintes sur le flux de contrôle, c'est-à-dire, sur l'ordre des opérations. La *dimension de données* (ou dimension *contrôle plus données*) introduit des opérateurs imposant aussi des restrictions sur les données, c'est-à-dire, sur les messages échangés entre les services.

Dans ce mémoire, nous considérons seulement cette deuxième dimension. Nous présentons une révision des opérateurs proposés dans [BHM06] en montrant leur sémantique dans le cadre de la théorie des traces et des graphes de dépendances ([BH08]).

Nous donnons la grammaire d'un programme PEWS :

$$P ::= (\textbf{var } X = \mathit{ArithExp})^*(\textbf{def } Z = \mathit{ArithExp})^*(\textbf{service } Y = S)^* S$$
$$S ::= O \mid Y \mid S + S \mid S^\star \mid \{S\} \mid [C]S \mid S <\|> S \mid S \ll\|\gg S \mid S \|\gg S \mid S \|> S \mid S.S$$

Un programme PEWS P est composé d'une séquence de :

- Définitions de variables : Une variable X est définie pour stocker la valeur résultant de l'évaluation d'une expression arithmétique (*ArithExp*). La valeur d'une variable peut être modifiée par le prédicat attribution (qui est toujours évalué *true*). Un programme PEWS nous permet aussi de définir (**def**) des variables spéciales Z qui seront évaluées à chaque fois que la variable est nécessaire pendant l'exécution. Il s'agit ici de la définition d'une fonction simple, qui a été introduite dans [BCHM05], mais que nous ne rappelons pas dans ce mémoire.
- Expressions des services : Une *path expression* S est associée à un identificateur Y.
- Une *path expression* : Cette expression représente le service défini par le programme. Remarquer que O correspond à des opérations (selon la définition donnée dans le document WSDL associé au programme) alors que C est un prédicat PEWS (une condition booléenne ou une attribution qui peut contenir des compteurs et des noms de variables). Les opérateurs pour la construction des *path expressions* sont similaires à ceux utilisés dans les expressions régulières. La grammaire S, ci-dessus, définit qu'une expression peut être une opération WSDL (O), un service définit en PEWS (Y), un choix non déterministe ($+$), la composition séquentielle (.) ou parallèle avec différentes restrictions sur les messages ($<\|>$, $\ll\|\gg$, $\|\gg$, $\|>$), les répétitions parallèles ($\{\ldots\}$) ou séquentielles ($\star$) ou l'exécution conditionnelle d'un service ($[C]S$).

7.4.1 Approche par la théorie des traces

Soit S la spécification d'un service. Un système de traces pour les messages $\mathcal{T}(S)$ est une paire $(\Sigma_S, Prefix(T(S)))$ où Σ_S est un alphabet de concurrence et $T(S)$ est un langage de traces.

Définition 7.4.1 : Alphabet de concurrence - Soit S un programme PEWS. Son alphabet de concurrence $\Sigma_S = (A_S, D_S)$ contient l'ensemble des messages associés à S par une spécification WSDL et la relation de dépendances D_S. Pour chaque S, Σ_S est défini de la façon suivante :

- $\Sigma_O = (A_O, D_O)$ où A_O est l'ensemble des messages associés à l'opération O par une spécification WSDL et D_O est définie comme suit :

$$D_O = \begin{cases} \{m\}^2 & \text{si } O \text{ est une opération } \textit{one-way } m? \text{ ou } \textit{ notification } m!. \\ \{m_1, m_2\}^2 & \text{si } O \text{ est une opération } \textit{request-response } m_1?.m_2! \text{ ou } \textit{solicit-response } m_2!.m_1?. \end{cases}$$

- $\Sigma_Y = \Sigma_{S_1}$ où "**service** $Y = S_1$" apparaît dans le programme PEWS. L'alphabet de concurrence d'un identificateur de service Y est l'alphabet de concurrence de la *path expression* correspondant.

- Étant donnés $\Sigma_{S_1} = (A_1, D_1)$ et $\Sigma_{S_2} = (A_2, D_2)$ avec $A_1 \cap A_2 = \emptyset$, alors

$$\Sigma_{S_1.S_2} = (A_1 \cup A_2, D_1 \cup D_2 \cup (A_1 \times A_2) \cup (A_2 \times A_1)).$$

  L'alphabet de concurrence d'une séquence de *path expressions* contient une relation de dépendances où tous les messages du premier service sont dépendants de tous les messages du deuxième service. Tous les messages du premier service sont distincts des messages du deuxième service.

- Étant donnés $\Sigma_{S_1} = (A_1, D_1)$ et $\Sigma_{S_2} = (A_2, D_2)$ avec $A_1 \cap A_2 = \emptyset$, alors :

$$\Sigma_{S_1+S_2} = (A_1 \cup A_2, D_1 \cup D_2).$$

L'alphabet d'un choix multiple est défini par l'union disjointe des alphabets concernés dans chaque opération. Néanmoins, si $\Sigma_{S_1} = \Sigma_{S_2}$ nous définissons $\Sigma_{S_1+S_2} = (A_1, D_1)$.

- Étant donnés $\Sigma_{S_1} = (A_1, D_1)$ et $\Sigma_{S_2} = (A_2, D_2)$ avec $((A_1 \cap A_2) \setminus \alpha) = \emptyset$, alors :

$$\Sigma_{S_1} \ll\|\gg_{S_2} = \Sigma_{S_1} <\|>_{S_2} = \Sigma_{S_1} \|\gg_{S_2} = \Sigma_{S_1} \|>_{S_2} = \Sigma_{S_1} \cup \Sigma_{S_2}$$

L'alphabet de concurrence des communications parallèles est défini comme l'union des alphabets concernés dans chaque opération.

- $\Sigma_{S^\star} = \Sigma_{\{S\}} = \Sigma_{[B]S} = \Sigma_S$
L'alphabet de concurrence d'une itération ou d'une expression conditionnelle est l'alphabet de concurrence de l'expression qui apparaît dans le corps de la spécification. □

Exemple 7.4.1 Soient $\Sigma_{S_1} = (\{m_1, m_2\}, ID)$ et $\Sigma_{S_2} = (\{m_3\}, \{m_3\}^2)$ deux alphabets de concurrence. Soit $S = S_1.S_2$ un service composé. Alors, $\Sigma_S = (\{m_1, m_2, m_3\}, \{m_1, m_3\}^2 \cup \{m_2, m_3\}^2\}$. Remarquer que m_1 et m_2 restent indépendants dans S. □

Définition 7.4.2 : Systèmes de traces - Soit S une spécification PEWS. Le système de traces $\mathcal{T}(S)$ est défini comme $\mathcal{T}(S) = (\Sigma_S, \mathit{Prefix}(T(S)))$ où $T(S)$ est défini de manière inductive sur les *path expressions* :

- $T(O) = \begin{cases} [m] & \text{si } O \text{ est une opération } \textit{one-way} \text{ ou } \textit{notification } m \\ [m_1.m_2] & \text{si } O \text{ est une opération } \textit{request-response} \text{ ou } \textit{solicit-response } m_1.m_2 \end{cases}$

L'ensemble des traces d'une opération unique contient une seule trace composée des messages définis par la spécification WSDL pour l'opération.

- $T(Y) = T(S_1)$ où "**service** $Y = S_1$" apparaît dans le programme PEWS. L'ensemble des traces d'un identificateur de service Y est l'ensemble de traces de la *path expression* correspondant.

- $T([C]S) = T(S) \cup \{[\epsilon]\}$. L'ensemble des traces d'une opération conditionnelle contient l'ensemble des traces de la sous expression (si la condition C est évaluée à **true**) et la trace vide (pour indiquer la possibilité de la condition C d'être évaluée à **false**).

- $T(S_1 + S_2) = T(S_1) \cup T(S_2)$. L'ensemble des traces d'un choix multiple est l'union (disjointe) des ensembles de traces concernés par l'opération.

- $T(S_1.S_2) = T(S_1).T(S_2)$. L'ensemble des traces d'une séquence est la concaténation des ensembles des traces concernés par l'opération.

- $T(S^\star) = \bigcup_{i \geq 0} T^i(S)$ où $T^0 = \{[\epsilon]\}$, $T^1 = T$ et pour $i > 1$, $T^i = T.T^{i-1}$. L'ensemble des traces d'une itération sur un même ensemble de dépendances D, contient la concaténation des traces individuelles de l'expression dans l'itération.

- $T(\{S\}) = \bigcup_{i \geq 0} T^i$ (avec $T = T(S)$) où $T^0 = \{[\epsilon]\}$, $T^1 = T$ et pour $i > 1$, $T^i = T \otimes T^{i-1}$. L'ensemble de traces d'une répétition parallèle est formé par le *shuffle* des ensembles individuels. Il n'y a pas de synchronisation entre deux occurrences d'un service S.

- Les langages de traces pour les différentes formes de communication en parallèle sont contraints par quelques conditions. Ci-dessous, soient α_t^{in} et α_t^{out} les alphabets d'entrée et de sortie d'une trace t et soit $T(S_1 \oslash S_2) = \bigcup_{1 \leq i \leq n, 1 \leq j \leq m} \mathcal{X}_{i,j}$ pour chaque opérateur parallèle $\oslash \in \{ <\|>, \|>, \ll\|\gg, \|\gg \}$. Pour chaque $t_1^i \in T(S_1)$ et $t_2^j \in T(S_2)$, la trace $\mathcal{X}_{i,j}$ est définie selon l'opérateur de la manière suivante :
  1. $T(S_1 <\|> S_2)$: Nous avons $\mathcal{X}_{i,j} = t_1^i \circledS_\alpha t_1^j$, pour $\alpha = (\alpha_{t_1^i}^{out} \cap \alpha_{t_2^j}^{in}) \cup (\alpha_{t_1^i}^{in} \cap \alpha_{t_2^j}^{out})$. Intuitivement, $S_1 <\|> S_2$ indique que les services S_1 et S_2 peuvent communiquer entre eux et avec d'autres services.
  2. $T(S_1 \|> S_2)$: Si $\alpha_{t_2^j}^{out} \cap \alpha_{t_1^i}^{in} = \emptyset$ alors $\mathcal{X}_{i,j} = t_1^i \circledS_\alpha t_1^j$, pour $\alpha = \alpha_{t_1^i}^{out} \cap \alpha_{t_2^j}^{in}$. Sinon $\mathcal{X}_{i,j} = [\emptyset]$. Intuitivement, $S_1 \|> S_2$ indique que *certains* messages de sortie d'un service S_1 peuvent

correspondre à *certains* messages d'entrée du service S_2. L'opérateur permet la communication de S_1 à S_2 mais interdit la communication de S_2 à S_1.

3. $T(S_1 \ll\|\gg S_2)$: Si $\alpha^{out}_{t^i_1} = \alpha^{in}_{t^j_2}$and $\alpha^{out}_{t^j_2} = \alpha^{in}_{t^i_1}$, alors $\mathcal{X}_{i,j} = t^i_1 \circledS_\alpha\ t^j_2$ pour $\alpha = \alpha^{out}_{t^i_1} \cup \alpha^{out}_{t^j_2}$. Sinon $\mathcal{X}_{i,j} = [\emptyset]$. La spécification $S_1 \ll\|\gg S_2$ définit un système clos : toutes les communications sont effectuées entre S_1 et S_2.
4. $T(S_1 \|\gg S_2)$: Si $\alpha^{out}_{t^i_1} = \alpha^{in}_{t^j_2}$and $\alpha^{out}_{t^j_2} \cap \alpha^{in}_{t^j_1} = \emptyset$ alors $\mathcal{X}_{i,j} = t^i_1 \circledS_\alpha\ t^j_2$, pour $\alpha = \alpha^{out}_{t^i_1}$. Sinon $\mathcal{X}_{i,j} = [\emptyset]$. Intuitivement, $S_1 \|\gg S_2$ indique que *tous* les messages de sorties d'un service S_1 correspondent à *tous* les messages d'entrée d'un service S_2. Aucune communication de S_2 à S_1 est possible.

Il est utile de préciser certains changements par rapport à la version de PEWS présentée dans [BHM06] :
(1) Dans ce mémoire, nous ne considérons pas les opérateurs séquentiels spéciaux, qui, en effet, utilisent l'opérateur de séquence et des opérateurs parallèles. Ils peuvent ainsi être obtenus à partir des opérateurs traités ci-dessus ([Ba08]).
(2) Pour des raisons pratiques (et de complexité) nous avons changé la sémantique de l'opérateur { } par rapport à celle présentée dans [BHM06].
(3) Les conditions sur les opérateurs parallèles sont exprimées en prenant en compte les alphabets spécifiques de chaque trace. Cela est une manière plus affinée et précise de les définir, car dans certains cas spécifiques, considérer l'alphabet du service globalement peut rendre difficile l'analyse des propriétés d'une composition.

Exemple 7.4.2 Soient les spécifications PEWS S_1, S_2, S_3 et S_4 telles que :

$\mathcal{T}_1 = \mathcal{T}(S_1) = (\{\{a,b,c\}, \{a,b\}^2 \cup \{b,c\}^2), \{[a?b!c?]\})$, $\mathcal{T}_2 = \mathcal{T}(S_2) = (\{\{d,e\}, \{d,e\}^2), \{[d!e?]\})$,
$\mathcal{T}_3 = \mathcal{T}(S_3) = (\{\{b,e,d\}, \{b,e\}^2 \cup \{b,d\}^2), \{[b?e!d?]\})$ et $\mathcal{T}_4 = \mathcal{T}(S_4) = (\{\{a,c\}, \{a,c\}^2), \{[a!c!]\})$.

Soit $S = ((S_1.S_2) <\|> S_3) \ll\|\gg S_4$. Nous avons $\mathcal{T}(S_1.S_2) = (\{\{a,b,c,d,e\}, \{a,b\}^2 \cup \{b,c\}^2 \cup \{d,e\}^2 \cup \{(a,d)\}^2 \cup \{(a,e)\}^2 \cup \{(b,d)\}^2 \cup \{(b,e)\}^2 \cup \{(c,d)\}^2 \cup \{(c,e)\}^2), [a?b!c?d!e?])$. Ensuite nous avons $\mathcal{T}((S_1.S_2) <\|> S_3) = (\{\{a,b,c,d,e\}, \{a,b\}^2 \cup \{b,c\}^2 \cup \{d,e\}^2 \cup \{(a,d)\}^2 \cup \{(a,e)\}^2 \cup \{(b,d)\}^2 \cup \{(b,e)\}^2 \cup \{(c,d)\}^2 \cup \{(c,e)\}^2), [a?b\natural c?d\natural e\natural])$. Finalement $\mathcal{T} = \mathcal{T}(S) = (\{\{a,b,c,d,e\}, \{a,b,c,d,e\}^2), \{[a\natural b\natural c\natural d\natural e\natural]\})$. □

7.4.2 Approche par les graphes de dépendances

Dans l'approche par les graphe de dépendances, les définitions sont équivalentes à celles de la section précédente ; seulement les TSDG remplacent les systèmes de traces.

Définition 7.4.3 Soit S une spécification PEWS. Le TSDG $\mathcal{G}(S)$ est défini de manière inductive sur les *path expressions* :

- PEWS spécifie des services simples à partir d'un ensemble d'opérations et des messages dans un fichier WSDL. Si O est une opération *one-way* ou *notification* alors $\mathcal{G}(O)$ est l'ensemble contenant seulement le graphe $G = (\{1\}, \emptyset, l)$ où $l(1) = a$. Si O est une opération *request-response* ou *solicit-response* alors $\mathcal{G}(O)$ est un ensemble contenant seulement le graphe $G = (\{1,2\}, \{(1,2)\}, l)$ où $l(1) = a$ et $l(2) = b$. Remarquer que a ou b représentent des messages avec des décorations ? ou !, c'est-à-dire, $a = m?$ ou $a = m!$.

- $\mathcal{G}(Y) = \mathcal{G}(S_1)$ où "**service** $Y = S_1$" apparaît dans le programme PEWS. Le TSDG d'un identificateur de service Y est le TSDG du *path expression* correspondant.

- $\mathcal{G}([C]S) = \mathcal{G}(S) \cup \mathcal{G}_\epsilon$ où $\mathcal{G}_\epsilon$ contient seulement G_ϵ. Le TSDG d'une opération conditionnelle contient le TSDG de la sous expression (si les conditions sont évaluées comme **true**) et $\mathcal{G}_\epsilon$ (pour indiquer la possibilité de la condition d'être évaluée comme **false**).

- $\mathcal{G}(S_1 + S_2) = \mathcal{G}(S_1) \cup \mathcal{G}(S_2)$. Le TSDG d'un choix multiple est l'union des TSDG des expressions concernées dans l'opération en considérant les graphes isomorphes comme un seul graphe dans $\mathcal{G}$.

  Dans la suite, si $\mathcal{G}(S_1) = \{G^1_1, \ldots, G^n_1\}$ and $\mathcal{G}(S_2) = \{G^1_2, \ldots, G^m_2\}$ alors $\mathcal{G}(S_1) \circ \mathcal{G}(S_2) = \{G^1_1 \circ G^1_2, G^1_1 \circ G^2_2, \ldots, G^n_1 \circ G^m_2\}$ où $\circ$ est un des opérateurs de l'ensemble $\{., \circledS, \otimes\}$.

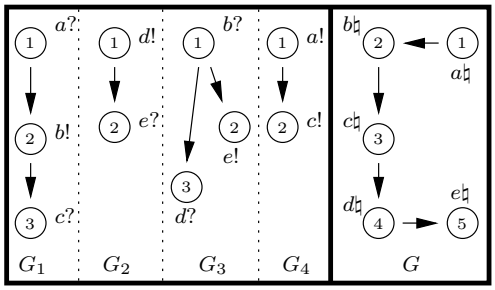
– $\mathcal{G}(S_1.S_2) = \mathcal{G}(S_1).\mathcal{G}(S_2)$. Le TSDG d'une séquence est la connexion (définition 7.3.5) de chaque graphe dans $\mathcal{G}(S_1)$ avec chaque graphe dans $\mathcal{G}(S_2)$.

– $\mathcal{G}(S^\star) = \bigcup_{i\geq 0} \mathcal{G}^i$ (avec $\mathcal{G} = \mathcal{G}(S)$) où $\mathcal{G}^0 = \{G_\epsilon\}$, $\mathcal{G}^1 = \mathcal{G}$ et pour $i > 1$, $\mathcal{G}^i = \mathcal{G} \odot \mathcal{G}^{i-1}$ (la concaténation de i copies de $\mathcal{G}$). Le TSDG d'une itération contient la concaténation des TSDG individuels de l'expression dans l'itération.

– $\mathcal{G}(\{S\}) = \bigcup_{i\geq 0} \mathcal{G}^i$ (avec $\mathcal{G} = \mathcal{G}(S)$) où $\mathcal{G}^0 = \{G_\epsilon\}$, $\mathcal{G}^1 = \mathcal{G}$ et pour $i > 1$, $\mathcal{G}^i = \mathcal{G} \otimes \mathcal{G}^{i-1}$.
Le TSDG d'une répétition parallèle est formé par le *shuffle* des TSDG individuels. Il n'y a pas de synchronisation entre deux occurrences d'un service S.

– Les langages de traces pour les différentes formes de communication en parallèle sont contraints par quelques conditions. Ci-dessous, soient α_G^{in} et α_G^{out} les alphabets d'entrée et de sortie d'un graphe de dépendances G et soit $\mathcal{G}(S_1 \oslash S_2) = \bigcup_{1\leq i\leq n, 1\leq j\leq m} \mathcal{X}_{i,j}$ pour chaque opérateur parallèle $\oslash \in \{ <\|> , \|> , \ll\|\gg , \|\gg \}$. De façon similaire à ce que nous avons pour les traces, pour chaque $G_1^i \in \mathcal{G}(S_1)$ et $G_2^j \in \mathcal{G}(S_2)$, le graphe $\mathcal{X}_{i,j}$ est défini selon l'opérateur de la manière suivante :

  1. $\mathcal{G}(S_1 <\|> S_2)$: Nous avons $\mathcal{X}_{i,j} = G_1^i \circledS_\alpha G_1^j$, pour $\alpha = (\alpha_{G_1^i}^{out} \cap \alpha_{G_2^j}^{in}) \cup (\alpha_{G_1^i}^{in} \cap \alpha_{G_2^j}^{out})$.
  2. $\mathcal{G}(S_1 \|> S_2)$: Si $\alpha_{G_2^j}^{out} \cap \alpha_{G_1^i}^{in} = \emptyset$ alors $\mathcal{X}_{i,j} = G_1^i \circledS_\alpha G_1^j$, pour $\alpha = \alpha_{G_1^i}^{out} \cap \alpha_{G_2^j}^{in}$. Sinon $\mathcal{X}_{i,j} = \{G_\emptyset\}$. L'opérateur permet la communication de S_1 à S_2 mais interdit la communication de S_2 à S_1.
  3. $\mathcal{G}(S_1 \ll\|\gg S_2)$: Si $\alpha_{G_1^i}^{out} = \alpha_{G_2^j}^{in}$ and $\alpha_{G_2^j}^{out} = \alpha_{G_1^i}^{in}$, alors $\mathcal{X}_{i,j} = G_1^i \circledS_\alpha G_2^j$ pour $\alpha = \alpha_{G_1^i}^{out} \cup \alpha_{G_2^j}^{out}$. Sinon $\mathcal{X}_{i,j} = \{G_\emptyset\}$. La spécification $S_1 \ll\|\gg S_2$ définit un système clos.
  4. $\mathcal{G}(S_1 \|\gg S_2)$: Si $\alpha_{G_1^i}^{out} = \alpha_{G_2^j}^{in}$ and $\alpha_{G_2^j}^{out} \cap \alpha_{G_1^j}^{in} = \emptyset$ alors $\mathcal{X}_{i,j} = G_1^i \circledS_\alpha G_2^j$, pour $\alpha = \alpha_{G_1^i}^{out}$. Sinon $\mathcal{X}_{i,j} = \{G_\emptyset\}$.



(a)

PEWS Specification :	Evaluation Time (s)
$((S_1.S_2) <\|> S_3) \ll\|\gg S_4$	3.10^{-2}
$(S_1.S_2) <\|> S_3$	2.10^{-2}
$S_1.S_2$	2.10^{-3}

(b)

FIG. 7.7: (a) L'évaluation de l'expression PEWS $S = ((S_1.S_2) <\|> S_3) \ll\|\gg S_4$, avec $\mathcal{G}(S) = \{G\}$ et $\mathcal{G}(S_i) = \{G_i\}$. (b) Temps pour vérifier la correction d'une expression.

Exemple 7.4.3 La figure 7.7 illustre la composition $S = ((S_1.S_2) <\|> S_3) \ll\|\gg S_4$ de l'exemple 7.4.2. Pour simplifier la figure, les arcs entre nœuds déjà connectés par un autre chemin n'ont pas été représentés. □

7.5 Propriétés

Notre but étant la construction d'une plate-forme d'aide à la conception des services web ; nous sommes intéressés par la vérification de certaines propriétés de nos compositions. Dans nos travaux nous avons considéré les tests suivants :

1. La *compatibilité* de services qui vérifie si deux ou plusieurs services forment un système clos.
2. La *correction* d'une composition qui assure la satisfaction d'une composition proposée par un utilisateur,

3. La *possibilité de substitution* d'un service qui assure que, dans une composition, un service donné peut être remplacé par un autre sans avoir à modifier sa collaboration avec d'autres services de la composition.

Dans la suite, nous présentons ces concepts du point de vue des traces ainsi que du point de vue des graphes, faisant le parallèle entre les deux formalismes.

Définition 7.5.1 Correction (*Soundness*) - Une expression PEWS S est *correcte* ssi $\mathcal{G}(S) \neq \{G_\emptyset\}$, c'est-à-dire, ssi $\mathcal{T}(S) \neq (\Sigma_S, \emptyset)$. □

Ainsi pour tester si la composition peut marcher, nous devons tester si le langage de traces (ou le TSDG) correspondant à la composition n'est pas vide. Remarquer que dans l'exemple 7.4.3, la composition est correcte.

Les notions de subsumption et d'équivalence sont importantes pour traiter les possibilités de substitution.

Définition 7.5.2 : *Subsumption* et équivalence des services - Soient S_1, S_2 deux expressions PEWS. Nous disons que S_1 est *subsumed* par S_2, noté $S_1 \preccurlyeq S_2$, ssi $\mathcal{G}(S_1) \subseteq \mathcal{G}(S_2)$, c'est-à-dire, les graphes abstraits de $\mathcal{G}(S_1)$ appartiennent à $\mathcal{G}(S_2)$.

En termes de traces ; en supposant que $\mathcal{T}(S_1) = (\Sigma_{S_1}, Prefix(T(S_1)))$ et $\mathcal{T}(S_2) = (\Sigma_{S_2}, Prefix(T(S_2)))$, nous disons que $S_1 \preccurlyeq S_2$ ssi $\Sigma_{S_1} \subseteq \Sigma_{S_2}$ et $T(S_1) \subseteq T(S_2)$.

Les spécifications S_1 et S_2 sont équivalentes, noté $S_1 \equiv S_2$, ssi $S_1 \preccurlyeq S_2$ et $S_2 \preccurlyeq S_1$. □

Le théorème suivant ([Ba08]) établit qu'un service S_1 peut être remplacé par un service S_2 si S_2 assure au moins toutes les conversations de S_1.

Théorème 7.5.1 Soient S, S_1, S_2 des expressions PEWS. Si $S_1 \preccurlyeq S_2$ et S_1 est une sous-expression de S, alors $S \preccurlyeq S[S_1/S_2]$, où $S[S_1/S_2]$ est une expression PEWS obtenue par la substitution de S_1 par S_2 dans S. □

Complexité et implémentation

Une implémentation partielle de l'éditeur PEWS a déjà été faite (UFRN, Brésil) et un prototype du programme de vérification de la composition est disponible sur la forme d'une API java [Ba08]. Cette implémentation a été faite sur l'approche graphe de dépendances, puisque l'implémentation des traces via des chaînes de caractères s'est avérée moins efficace.

En supposant qu'une bibliothèque de services est disponible, contenant des spécifications PEWS ou des TSDG déjà calculés, la version actuelle de notre API fait la vérification de la correction d'une composition S construite à partir des services dans la bibliothèque. La figure 7.7 montre les temps d'exécution de cette procédure pour différentes expressions PEWS. Dans [BH08] nous trouvons les détails sur l'algorithme de synchronisation.

Il est important de remarquer que l'implémentation de la synchronisation dans l'approche de traces (c'est-à-dire, avec des mots) est moins efficace. En effet, la synchronisation de deux traces, dans le pire de cas, est $O(n!)$ où n est la taille de la trace. Cela se justifie car nous avons besoin de vérifier toutes les permutations possibles (des symboles indépendants) pour trouver la bonne solution. Par exemple, dans l'exemple 7.4.3, la trace $t = [a?b!c?d!e?]$ est représentée par $G_1.G_2$ et $t_1 = [b?e!d?]$ est représentée par G_3. Pour synchroniser t et t_1, il est nécessaire de prendre en compte le fait que $t_1 = \{b?e!d?, b?d?e!\}$ pour ensuite faire la synchronisation en considérant le mot $b?d?e!$ (le cas qui nous permet un nombre maximal de synchronisation). Les graphes de dépendances coupent l'ordre linéaire imposé par les chaînes de caractères et permettent des implémentations plus efficaces.

La vérification de la correction d'une composition PEWS est faite en temps $O(n^m \times t)$ où n est le nombre de graphes de dépendances dans chaque $\mathcal{G}(S_i)$, m est le nombre d'opérateurs PEWS et t est la complexité de la synchronisation ou de la connexion. Il est important de remarquer que si nous utilisons une bibliothèque de services stockant des TSDG déjà calculé, la valeur de m est souvent petite. Dans les exemples de la figure 7.7, le temps est calculé pour des exécutions sur un PC Intel Pentium(R) M Processor 1.70 GHz, 512 Mo RAM., avec $n = 1$.

7.6 Travaux liés

Dans le cadre de la composition de services, d'un coté nous trouvons différentes propositions qui ont comme but de devenir des langages standards pour l'industrie, d'un autre coté nous trouvons des propositions des formalismes pour la description et la vérification des propriétés des services ([MM04]). Dans le premier groupe, le langage le plus cité pour spécifier la composition des services est BPEL [ACD+03] (Business Process Execution Language). Un programme BPEL définit des services web en centralisant la description de la composition dans une seule unité, c'est-à-dire, BPEL est un langage d'orchestration. Même si BPEL est très utilisé, ce langage offre des possibilités de vérification très limitées ; motivant ainsi différentes propositions qui visent à améliorer cette situation. Par exemple, dans [Ake04] nous trouvons un outil pour l'analyse statique de BPEL alors que [VvdA05] décrit certains constructeurs BPEL en utilisant des réseaux de Petri. Néanmoins, à notre connaissance, il n'existe pas encore d'outils pour vérifier la correction d'un programme BPEL [MM04], ni pour analyser les possibilités de substitution et de compatibilité des services.

Plusieurs propositions récentes admettent l'importance d'une analyse formelle des protocoles des services ([BCPT05, BCH05, BFHS03, FBS04, SCZ04]). Le traitement formel des interactions de services web peut être exploité dans la composition automatique que nous ne considérons pas dans nos travaux. Nous nous plaçons dans une plate-forme d'aide à l'utilisateur : L'utilisateur connaît très bien son domaine d'application et possède des bases fondamentales pour utiliser un langage de spécification comme PEWS. Il peut ensuite demander une vérification pour tester si sa composition peut fonctionner, c'est-à-dire, si elle est correcte. Notre choix est motivé par les besoins des professionnels voulant composer des services web à partir d'une bibliothèque de services mise à disposition par différents partenaires. Contrairement à nous, différents travaux considèrent le problème de la composition automatique, comme par exemple [CFB04, PBB+04, PTBM05].

Nous proposons PEWS comme langage pour la spécification de interfaces des services web et nous utilisons la théorie des traces et les graphes de dépendances pour analyser les possibles interactions entre services. Des travaux similaires, proposant d'autres formalismes, sont présentés, par exemple, dans [dAH01, BCH05, BCPT05, HB03, SBS04]. Il est intéressant de remarquer que plusieurs problèmes dans le domaine de services web trouvent une correspondance dans le domaine des composants de logiciels (voir [dAH01] comme exemple). Les différences plus marquantes entre la plupart de ces approches et la nôtre sont : la définition d'une composition comme étant seulement un service clos, sans prendre en compte d'autres possibilités ; les tests de corrections et de remplacement, définis dans une approche "pessimiste", c'est-à-dire, fondés sur la vérification que toutes les possibilités d'exécution mènent à un résultat correct, et non seulement sur l'existence d'*une* possibilité d'exécution correcte.

En effet, comme dans [dAH01], notre travail se place dans une approche "optimiste", c'est-à-dire, nos algorithmes répondent favorablement à une composition proposée par l'utilisateur s'il existe *une possibilité* d'exécution correcte. Cette prise de position nous permet d'avoir des algorithmes plus légers et efficaces par rapport aux propositions de l'approche "pessimiste" adopté pour plusieurs travaux.

Bien que les approches décrites dans [BCH05, SBS04] soient pessimistes, certains points en commun avec la nôtre sont à remarquer. Dans [SBS04] l'algèbre de processus est proposée comme formalisme pour décrire, raisonner et faire des vérifications sur les interfaces des services web. L'algèbre de processus est aussi le formalisme adopté dans [MBSR06, Yeu06], entre autres. Dans [BCH05], le comportement d'un service est spécifié par une interface de protocole qui décrit l'exécution des actions, utilisant les automates d'états finis. Les deux travaux considèrent les problèmes d'équivalence entre services et de la correction d'une composition. Néanmoins, étant dans une approche pessimiste la bissimulation est la méthode utilisée pour vérifier la possibilité de remplacement, ce qui rend les algorithmes plus complexes que le nôtre.

Dans [BCPT05] nous trouvons une spécification comportementale de service nommée *timed protocol*. La modélisation utilise des automates d'état finis où les transitions sont activées par des messages ou par des *timeouts*. Dans [BCPT05] les traces sont des suites d'évènements alors que dans notre approche une trace est une classe d'équivalence sur l'ensemble de toutes les chaînes. Dans PEWS les prédicats englobant le temps sont possibles ([BCHM05]) et nous pouvons ainsi exprimer les *timeouts*. La notion de compatibilité de [BCPT05] est incluse dans notre définition de correction. Finalement, même si dans [BCPT05] différents types de remplacements sont définis cette notion reste similaire à la notre.

Dans [HB03] une algèbre pour la composition de services web, fondée sur les réseaux de Petri, est proposé. Certains opérateurs proposés trouvent leur équivalents en PEWS, mais ceux qui imposent des restrictions sur la communication ne sont pas pris en compte dans [HB03] où le flux de contrôle reste la préoccupation principale. Ce travail est similaire au nôtre car, à partir d'une composition exprimée en réseaux de Petri, il est possible d'effectuer des tests sur la correction de la composition ainsi qu'évaluer certaines possibilités de substitution. Par contre, dans [KP06] nous trouvons une approche où les flux des données sont aussi considérés. Le concept de traces semble pouvoir simuler les *runs* utilisés par les auteurs ; mais dans nos travaux nous n'avons pas considéré la correction par rapport à une condition donnée (exprimée avec la logique temporelle LTL dans [KP06]).

7.7 Conclusions

L'utilisation des *path expressions* comme base du langage PEWS ainsi que l'emploi de la théorie des traces pour le raisonnement sur la composition des services web est l'originalité de notre approche.

PEWS a comme avantage le fait d'être un langage formel simple, similaire aux expressions régulières. La traduction d'une spécification PEWS dans la théorie des traces ou dans les TSDG est complètement transparente pour l'utilisateur. Les vérifications statiques offertes par notre approche pour la correction et la substitution peuvent être implementés par des algorithmes ayant une bonne performance dans la pratique (selon nos premiers tests).

La comparaison entre les différentes propositions dans notre contexte doit prendre en compte le pouvoir d'expressions des formalismes considérés. PEWS est naturellement adapté au modèle WSDL ; puisque un programme PEWS peut être vu dans la *dimension contrôle* ou dans la dimension *contrôle plus données*. La dimension contrôle de PEWS peut exprimer la plupart des motifs de flux de contrôle, comme montré dans [MP06] : PEWS peut exprimer plus de motifs que les langages BPEL, XLANG, WSFL, BPML, WSCI.

Chapitre 8

Conclusions et perspectives

Des difficultés sont à prévoir lors de la mise à jour d'une base de données ayant des contraintes à respecter, car, des données originalement cohérentes par rapport aux contraintes peuvent devenir incohérentes suite aux mises à jour. Depuis mon doctorat, ce thème est présent dans mes travaux, même s'il est étudié dans des contextes très différents. En effet, il est possible de classer mes travaux de recherche dans quatre contextes principaux, à savoir :

1. Les aspects dynamiques des bases de données déductives.
2. La maintenance incrémentale des entrepôts de données via des opérateurs de la logique temporelle.
3. Les mises à jour et la validation incrémentale des bases de données XML.
4. Les langages pour la définition, la composition et la manipulation des services web.

Alors que le thème 1 est le sujet de ma thèse de doctorat, les thèmes suivants représentent des recherches entamées à différentes périodes après mon doctorat. Ce mémoire décrit mes recherches dans les thèmes 3 et 4.

Ainsi, dans le domaine XML nous avons proposé différentes politiques pour la préservation de la cohérence de la base, à savoir, le simple rejet de la mise à jour, la modification des données stockées et même, dans certaines circonstances spéciales, le changement des contraintes imposées sur les données. Les contraintes de schéma et d'intégrité ont été considérées mais de façon indépendante. Nos algorithmes de validation sont fondés sur l'exécution d'un automate d'arbre, mais la validation via programme logique (datalog monadique) a aussi été abordée.

Des prototypes pour la validation incrémentale par rapport aux contraintes d'intégrité et de type ainsi que pour l'évolution de schéma ont été implémentés et il serait intéressant de mettre en place une plate-forme où les fonctionnalités proposées par nos travaux soient disponibles.

Dans le domaine des services web, nos travaux sont plus récents, mais illustrent assez bien notre objectif d'introduire un langage formel mais simple, sémantiquement bien défini, pour la spécification de services simples ou composés. Des vérifications statiques concernant la correction et la substitution des services d'une composition ont été discutées et implémentées.

Différentes perspectives dans les deux domaines, XML et services web, sont envisageables. Dans ce chapitre, je présente les perspectives XML que nous avons définies comme but dans le projet WebDex (projet ANR soumis en 2007). Ces objectifs concernent la considération des types non réguliers ainsi que les vérifications statiques de validité. Du coté des services web, nous comptons continuer l'étude des propriétés de PEWS ainsi que la conception des outils nécessaires pour la construction de la plate-forme d'aide à la spécification et manipulation des services web.

8.1 Du coté de XML

Nouvelles primitives de mises à jour et les contraintes de type au delà des grammaires régulières

Les opérations de mises à jour considérées dans nos travaux (définition 3.2.1) pourraient être classées comme des mises à jour "horizontales" en opposition à d'autres mises à jour que nous classerons comme les mises à jours "verticales". Pour illustrer la différence, nous considérons un arbre $a(t_1, t_2, t_3)$ où a est un label et chaque t_i est un sous arbre. Un exemple de mise à jour "horizontale" serait le remplacement d'un sous arbre par un autre, en transformant par exemple $a(t_1, t_2, t_3)$ en $a(t_1, t_{new}, t_3)$. Un exemple d'une mise à jour verticale serait l'insertion ou la suppression d'un niveau de la hiérarchie dans l'arbre, comme la transformation de l'arbre $a(t_1, t_2, t_3)$ en $a(k(t_1, t_2), t_3)$ et vice-versa (où k est un label). L'intérêt pour des mises à jour verticales peut exister, même en sachant que il est possible de les simuler par une suite de mises à jour horizontales (dans l'exemple précédent, par la suppression des sous arbres t_1 et t_2 et l'insertion de $k(t_1, t_2)$).

Nous pouvons argumenter que la restriction des mises à jour aux opérations "horizontales" s'explique en partie par le fait que les contraintes XML imposent une vison par niveau, et donc une vision "horizontale". Par exemple, les contraintes de schéma capables de restreindre la structure d'un document correspondent à des grammaires d'arbre avec des règles $A \rightarrow a[E]$ imposant des contraintes sur les fils d'un nœud de l'arbre (qui doivent respecter E).

Aucun langage de schéma actuel n'impose des contraintes qu'on pourrait considérer comme "verticales". Par exemple, supposons une contrainte $C \rightarrow c(A^{|n}.B^{|n})$ qui imposerait à un nœud d'avoir l'étiquette c, un premier fils A ayant lui même n fils A et un deuxième fils B ayant lui même le même nombre de fils que son frère. Ce type de contrainte ne correspond pas à des grammaires régulières d'arbre. Des formalismes plus puissants sont nécessaires pour les exprimer et cet aspect concerne une de nos perspectives de recherche.

Nous sommes ainsi intéressés par la représentation des schémas définissant des langages d'arbres non réguliers. Certains travaux ont pris en compte des contraintes horizontales non régulières, c'est-à-dire, des contraintes (plus puissantes que les expressions régulières) imposées sur les fils d'un nœud. Par exemple les contraintes de comptage ont été étudiées dans [DL02, SSM03] alors que dans [GT07] des contraintes plus générales (de type algébrique sont introduites. Des travaux concernant les contraintes synchronisées sur les grammaires de n-uplets d'arbres ([LS06, RCC05]) montrent comment exprimer des contraintes synchronisées verticales pour les arbres d'arité fixe.

Dans ce cadre, et en considérant l'évolution des schémas, un premier objectif peut être l'extension de l'approche $\texttt{dGREC}^{Dat}$ à un contexte non régulier. Par exemple, nous supposons que les contraintes imposées sur les fils d'un nœud sont établies par une grammaire hors contexte (CFG) et nous proposons d'étendre l'approche $\texttt{dGREC}^{Dat}$. Le but est de considérer le même problème traité par [dHM07] dans un nouveau cadre où les expressions régulières sont remplacées par les CFG et les programmes datalog sont remplacés par les programmes logiques. L'idée générale de cette approche est donc de caractériser le programme logique qui représente un type non régulier et de l'utiliser comme un validateur, de la même façon que nous avons utilisé le validateur datalog. Dans le cadre des arbres d'arité fixe, il existe une hiérarchie de langages d'arbre et les grammaires correspondantes (parfois exprimées par des programmes logiques) ([LS06]). Le but ici est d'étendre ces travaux aux arbres d'arité variables pour proposer un algorithme d'évolution de schémas capable de traiter des schémas (types) non réguliers.

Le traitement des contraintes plus puissantes (sur la verticale) impose l'ajout de nouvelles primitives de mise à jour, notamment celles permettant l'ajout ou la suppression d'un niveau de la hiérarchie. Par exemple, nous pouvons proposer $createFather(p_{beg}, p_{end}, lab)$ qui introduit un nœud avec label lab à la position p_{beg} prenant comme fils les sous-arbre enracinés de p_{beg} à p_{end}. Autrement dit, le premier fils du nœud p_{beg} dans le nouvel arbre correspond au sous arbre de racine p_{beg} de l'arbre original, le deuxième au frère de droite de p_{beg} (dans l'arbre original) et ainsi de suite jusqu'au dernier fils qui correspondra au sous arbre de racine p_{end} (dans l'arbre original). C'est donc l'opération $createFather(0, 1, k)$ qui permettrais de transformer l'arbre $a(t_1, t_2, t_3)$ en $a(k(t_1, t_2), t_3)$. Une opération inverse serait, par exemple, $dismissFather(p)$. L'introduction de ces opérations implique une révision des résultats obtenus dans nos travaux jusqu'à maintenant.

En ce qui concerne l'évolution de schéma, d'autres politiques pourraient aussi être prises en compte

comme par exemple, l'évolution des schémas non conservatrice (via des opérations de modification du schéma). Une telle opération pourrait être envisagée et mise en place en accord avec une validation incrémentale (concernant seulement les parties modifiées, comme dans [GMR05]) et la correction des documents devenus non valides par rapport au nouveau schéma.

Un langage de mises à jour

Comme je l'ai déjà signalé dans ce mémoire, nous avons jusqu'à maintenant considéré seulement un ensemble de primitives de mises à jour, sans prendre en compte leur intégration dans un langage de requête. En effet, jusqu'à maintenant, nous avons considéré qu'un utilisateur manipule un document XML (ou une partie d'un document) via un éditeur qui lui donne une vision de l'arbre XML et fournit des possibilités de modifications : l'utilisateur peut signaler les positions (dans l'arbre XML) où il souhaiterait effectuer des insertions, des suppressions ou des remplacements. Ces changements seraient appliqués aux documents, seulement suite à une opération de `committ`, à sa demande. Malheureusement l'implémentation de cette interface n'a pas encore été faite (seulement des prototypes ont été présentés par des étudiants de master).

La définition des mises à jour et des contraintes via un langage de requête ouvre une autre perspective à nos travaux. L'utilisation des requêtes arbre pourrait bien s'adapter aux cadre considéré jusqu'à maintenant, permettant à l'utilisateur d'avoir une vison graphique des mises à jour et des contraintes à établir. Néanmoins, d'autres possibilités restent à exploiter et pour cela, nous devons prendre en compte les langages déjà proposés pour la spécification des mises à jour et l'intégration de nos primitives à ces langages.

L'utilisation d'un langage de mise à jour nous dirige vers la recherche de mécanismes pour des analyses statiques. Par exemple, soient une contrainte d'intégrité et une mise à jour, exprimées, par exemple, par une requête arbre. L'analyse statique de ce deux "requêtes" devrait pouvoir nous dire si la mise à jour peut violer cette contrainte. Cette perspective nous approche des travaux déjà mentionnés ici, comme par exemple [GI06, GRS07]. Jusqu'à maintenant toutes nos vérifications de validité des contraintes sont dynamiques.

Une autre perspective de recherche que je considère intéressante est la transposition des règles de mises à jour proposées dans ma thèse au contexte XML. Étant données des mises à jour, nous voulons engendrer, à partir des règles de mise à jour, une liste non contradictoire, contenant les mises à jour données et déduites par les règles de mises à jour. Plusieurs questions peuvent être analysées dans ce cadre, notamment l'analyse de la cohérence des règles de mise à jour ainsi que leur interaction avec les autres contraintes de type et d'intégrité.

8.2 Du coté de PEWS

L'étude des propriétés de notre formalisme, comme l'équivalence entre différentes expressions PEWS, peut donner des résultats intéressants en ce qui concerne l'optimisation des spécifications. Cette étude a déjà commencé, dans le cadre de la thèse de Cheikh Ba ainsi que dans des travaux de fin d'étude des étudiants de Martin Musicante. Ainsi, l'équivalence $S_1 \gg S_2 \equiv (S_1.S_2) <\|> B$, où $\gg$ est un opérateur séquentiel avec des restrictions sur les données ([BHM06]) et B est un service "buffeur", a été prouvée dans [dS06]. La possibilité d'avoir aussi des algorithmes qui implémentent une approche pessimiste est à étudier.

Une comparaison plus détaillée de notre approche avec d'autres formalismes proposés dans la littérature mérite considération, en particulier la définition de nos limites ou avantages face aux approches fondées sur les réseaux de Petri ainsi que les rapports des spécifications PEWS avec les contrats de PiDuce[1]. Cette étude est sans doute laborieuse, mais peut ouvrir des portes à différentes perspectives.

Dans nos travaux actuels les synchronisations entre messages sont traitées de manière abstraite, c'est-à-dire, nous ne précisons pas comment un message d'entrée $m?$ correspond au message de sortie $m!$. Supposons par exemple, qu'un service S_1 attend comme entrée un message contenant le *prix* d'un produit, alors qu'un service S_2 envoie un message concernant le *coût* d'un produit. Pourront ils communiquer ? Il nous faut donc établir cette correspondance. L'utilisation des systèmes de types, comme dans les langages de programmation, est envisageable. Une autre option serait de prendre comme base une ontologie

[1] `http ://www.cs.unibo.it/PiDuce/`

spécifique. Nous n'avons pas encore exploité ces directions.

Différentes extensions de la plate-forme d'aide à la conception de services web sont envisageables. L'évolution efficace des spécifications PEWS par le remplacement, l'ajout ou la suppression des services composants. L'idée serait de pouvoir mettre à jour une composition, en remplaçant certains services par d'autres plus récents ou disponibles. Une perspective beaucoup plus ambitieuse serait de rendre la composition dynamique, comme suggéré dans [Ama03] : au lieu de spécifier (manuellement) le service à utiliser dans une composition, la spécification serait faite par des requêtes définissant le service souhaité. Au moment de l'utilisation du service composé les services composants seraient cherchés parmi les services disponibles capables de répondre aux requêtes.

Bibliographie

[AAF+02] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, and Sinisa Zimek. Web service choreography interface. Available at http ://www.w3.org/TR/wsci/, 2002.

[ABH+04] M. A. Abrão, B. Bouchou, M. Halfeld Ferrari, D. Laurent, and M. A. Musicante. Incremental constraint checking for XML documents. In *XSym*, number 3186 in LNCS, pages 112–127, 2004.

[ABPRT04] Daniel Austin, Abbie Barbir, Ed Peters, and Steve Ross-Talbot. Web services choreography requirements. Available at http ://www.w3.org/TR/2004/WD-ws-chor-reqs-20040311/, March 2004. W3C Working Draft.

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web - From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Didier Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weeranwarana. Bussiness process execution language for web services. Available at http ://www-128.ibm.com/developerworks/library/specification/ws-bpel/, 2003.

[AFL02] M. Arenas, W. Fan, and L. Libkin. On verifying consistency of XML specifications. In *PODS - Symposium on Principles of Database System*. ACM Press, 2002.

[AFL05] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. Consistency of XML specifications. In *Inconsistency Tolerance*, pages 15–41, 2005.

[AH00] Sandra de Amo and Mirian Halfeld Ferrari Alves. Efficient maintenance of temporal data warehouses. In *International Database Engeneering and Applications Symposium (IDEAS)*, Japan, September 2000.

[AH04] Sandra de Amo and Mirian Halfeld Ferrari Alves. Incremental maintenance of data warehouses based on past temporal logic operators. *Journal of Universal Computer Science*, 10(9) :1035–1064, 2004.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

[Ake04] D. H. Akehurst. Validating BPEL Specifications using OCL. Technical Report 15-04, University of Kent at Canterbury, August 2004.

[AL02] M. Arenas and L. Libkin. A normal form for XML documents. In *PODS - Symposium on Principles of Database System*. ACM Press, 2002.

[Ama03] Bernd Amann. Du partage centralisé de ressources web à l'échange de documents intensionnels. Technical report, Université Pierre et Marie Curie - Paris 6, Habilitation à diriger les recherches, 2003.

[And79] Sten Andler. Predicate path expressions. In *Sixth Annual ACM Symposium on Principles of Programming Languages (6th POPL'79)*, pages 226–236, 1979.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley, 1988.

[ATKB03] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1) :5–51, 2003.

[Ba08] Cheikh Ba. *Composing web services with PEWS*. PhD thesis, LI, Université François Rabelais de Tours (In preparation, in french), 2008.

[BBFV05] Michael Benedikt, Angela Bonifati, Sergio Flesca, and Avinash Vyas. Adding updates to XQuery : Semantics, optimization, and static analysis. In *XIME-P*, 2005.

[BBG+02] M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng. Automated update management for XML integrity constraints. In *Program Language Technologies for XML (PLANX02)*, 2002.

[BCCN06] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based xml projection. In *Proceedings of the VLDB Conference*, 2006.

[BCG+03] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services. Technical Report 22-2003, Dipartimento di Informatica e Sistemistica, Universita di Roma La Sapienza, Roma, Italy, 2003.

[BCH05] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web service interfaces. *International World Wide Web Conference Committe (IW3C2)*, 2005.

[BCH+07] Béatrice Bouchou, Ahmed Cheriat, Mírian Halfeld Ferrari, Dominique Laurent, Maria-Adrina Lima, and M. Musicante. Efficient constraint validation for updated XML databases. *Informatica*, 31(3) :285–310, 2007. Print edition ISSN : 0350-5596, `http ://ai.ijs.si/informatica/`.

[BCHM05] C. Ba, M. Carrero, M. Halfeld Ferrari, and M. Musicante. PEWS : A new language for building web service interfaces. *Journal of Universal Computer Science*, 11(7) :1215–1233, July 2005. http ://www.jucs.org/jucs_11_7/pews_a_new_language.

[BCHS06a] Béatrice Bouchou, Ahmed Cheriat, Mirian Halfeld Ferrari Alves, and Agata Savary. Integrating correction into incremental validation. In *BDA*, 2006.

[BCHS06b] Béatrice Bouchou, Ahmed Cheriat, Mírian Halfeld Ferrari, and Agata Savary. XML document correction : Incremental approach activated by schema validation. In *International Database Engeneering and Applications Symposium (IDEAS)*, 2006.

[BCPT05] Boualem Benatallah, Fabio Casati, Julien Ponge, and Farouk Toumani. Compatibility and replaceability analysis for timed web services protocols. *Bases de données avancées (BDA)*, 2005.

[BDF+01] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. C. Tan. Keys for XML. In *WWW10, May 2-5*, 2001.

[BDF+03] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. Reasoning about keys for XML. *Inf. Syst.*, 28(8) :1037–1063, 2003.

[BDH+04a] B. Bouchou, D. Duarte, M. Halfeld Ferrari, D. Laurent, and M. A. Musicante. Conservative extensions of regular languages. In *XXIV International Conference of the Chilean Computer Science Society (SCCC)*, 2004.

[BDH+04b] B. Bouchou, D. Duarte, M. Halfeld Ferrari, D. Laurent, and M. A. Musicante. Schema evolution for XML : A consistency-preserving approach. In *Mathematical Foundations of Computer Science (MFCS)*, number 3153 in Lecture Notes in Computer Science, pages 876–888. Springer-Verlag, August 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.

[BDHM07] B. Bouchou, D. Duarte, M. Halfeld Ferrari, and M. A. Musicante. Extending an XML type using updated data. *Submitted to an international journal*, 2007.

[BFHS03] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification : a new approach to design and analysis of e-service composition. *Proc. WWW, pp. 403-410. ACM*, 2003.

[BFK03] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. In *ICDT*, pages 79–95, 2003.

[BH03] B. Bouchou and M. Halfeld Ferrari Alves. Updates and incremental validation of XML documents. In Springer, editor, *The 9th International Workshop on Data Base Programming Languages (DBPL)*, number 2921 in LNCS, 2003.

[BH08] Cheikh Ba and Mirian Halfeld Ferrari. Dependence graphs for verifications of web service compositions with PEWS. In *Proceeding of the 23rd ACM Symposium on Applied Computing (SAC - SIGAPP), special track on Web Technologies (To appear)*, 2008.

[BHL07] Béatrice Bouchou, Mírian Halfeld Ferrari, and Maria-Adrina Lima. Contraintes d'intégrité pour XML : une visite guidée par une syntaxe homogène. *Submitted to a french journal*, 2007.

[BHM03] B. Bouchou, M. Halfeld Ferrari Alves, and M. A. Musicante. Tree automata to verify key constraints. In *Web and Databases (WebDB)*, San Diego, CA, USA, June 2003.

[BHM05] Cheikh Ba, Mirian Halfeld Ferrari, and Martin Musicante. Building web services interfaces using predicate path expressions. In *Proceedings of SBLP 2005. IX Brazilian Symposium on Programming Languages*, pages 147–160, Recife - Brazil, May 2005. Brasilian Computer Science Society, University of Pernambuco.

[BHM06] Cheikh Ba, Mirian Halfeld Ferrari, and Martin A. Musicante. Composing web services with PEWS : A trace-theoretical approach. In *IEEE European Conference on Web Services (ECOWS)*, pages 65–74, 2006.

[BK07] Michael Benedikt and Christoph Koch. XPath leashed. *ACM Computing Surveys*, 2007. To appear.

[BMW01] A. Brüggeman-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over non-ranked alphabets. Technical Report HKUST TCSC 2001 05, Hong Kong Univ. of Science and Technology Computer Science Center (available at `http ://www.cs.ust.hk/tcsc/RR/2001Â-05.ps.gz`), 2001.

[BPV04] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of xml documents. *ACM Trans. Database Syst.*, 29(4) :710–751, 2004.

[BW98a] A. Brüggeman-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2) :182–206, 1998.

[BW98b] A. Brüggeman-Klein and D. Wood. Regular tree languages over non-ranked alphabets. unpublished manuscript, 1998.

[BW04] A. Brüggeman-Klein and D. Wood. Balanced context-free grammars,hedge grammars and pushdown caterpillar automata. In *Extreme Markup Languages*, 2004.

[Büc60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math Logik Grundlag. Math.*, 6 :66–92, 1960.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. Availabre at http ://www.w3.org/TR/wsdl, 2001.

[CDG+02] 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 (new version 2002).

[CDZ02] Y. Chen, S. Davidson, and Y. Zheng. Validating constraints in XML. Technical Report MS-CIS-02-03, Department of Computer and Information Science, University of Pennsylvania, 2002.

[CFB04] Ion Constantinescu, Boi Faltings, and Walter Binder. Large scale, type-compatible service composition. In *ICWS*, pages 506–513. IEEE Computer Society, 2004.

[CFR] D. Chamberlin, D. Florescu, and J. Robie. XQuery update facility, W3C Working Draft 11 July 2006. Available at http ://www.w3.org/TR/2006/WD-xqupdate-20060711/.

[CGT90] S. Ceri, G.Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

[Che06] Ahmed Cheriat. *Une méthode de correction de la structure de documents XML dans le cadre d'une validation incrémentale*. PhD thesis, LI, Université François Rabelais de Tours, 2006.

[Cho92] J. Chomicki. History-less checking of dynamic integrity constraints. In *IEEE - Int. Conf. on Data Engineering*, pages 557–564, February 1992.

[CSBH05] Ahmed Cheriat, Agata Savary, Béatrice Bouchou, and Mírian Halfeld Ferrari. Incremental string correction : Towards correction of XML documents. In *Proceedings of the of the Prague Stringology Conference (PSC'05)*, 2005.

[CZ00] Pascal Caron and Djelloul Ziadi. Characterization of Glushkov automata. *Theor. Comput. Sci. (TCS)*, 233(1-2) :75–90, 2000.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[Deb05] Denis Debarbieux. *Modélisation et requêtes des documents semi-structurés : exploitation de la structure de graphe*. Phd thesis, Université des Sciences et Technologies de Lille, 2005.

[dHM07] Robson da Luz, Mírian Halfeld Ferrari, and Martin A. Musicante. Regular expression transformations to extend regular languages (with application to a datalog XML schema validator). *Journal of Algorithms (Special Issue)*, 62(3-4) :148–167, 2007.

[DL02] Silvano Dal Zilio and Denis Lugiez. XML Schema, Tree Logic and Sheaves Automata. Technical report, INRIA - Sophia Antipolis , Equipe : MIMOSA, Novembre 2002.

[dL07] Robson J. P. da Luz. Um ambiente para o aprendizado de gramáticas de Árvore. Master's thesis, Universidade Federal do Paraná, 2007. (In preparation, in portuguese).

[Don70] John Doner. Tree acceptors and some of their applications. *J. Comput. Syst. Sci.*, 5(4), 1970.

[dS06] André Luís de Souza Brito. Proving properties in PEWS. Technical report, Universidade Federal do Rio Grande do Norte (In Portuguese), 2006.

[Dua05] Denio Duarte. *Une méthode pour l'évolution de schémas XML préservant la validité des documents*. Phd thesis, Université François-Rabelais de Tours, LI/Campus Blois, 2005.

[Fan05] Wenfei Fan. XML constraints : Specification, analysis, and applications. In *DEXA Workshops*, pages 805–809, 2005.

[FBS04] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. *Proc. ICWS, pp. 96-103. IEEE*, 2004.

[FG02] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. In *LICS '02 : Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 215–224. IEEE Computer Society, 2002.

[FKS01] W. Fan, G. M. Kuper, and J. Siméon. A unified constraint model for XML. In *WWW10, May 2-5*, 2001.

[FS03] W. Fan and J. Simeon. Integrity constraints for XML. *Journal of Computer and System Science (JCSS)*, 66(1) :254–291, 2003.

[FUV83] R. Fagin, J. Ullman, and M. Y. Vardi. On the semantics of updates in databases. In *PODS - Symposium on Principles of Database System*. ACM Press, 1983.

[FV83] Ronald Fagin and Moshe Y. Vardi. Armstrong databases for functional and inclusion dependencies. *Information Processing Letters*, 16 :13–19, 1983.

[GI06] Françoise Gire and Hicham Idabal. Automate d'arbre pour l'interrogation incrémentale des bases de données semi-structurées. 2006.

[GK02] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. In *PODS'02 : Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 17–28, New York, NY, USA, 2002. ACM Press.

[GKP03] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The complexity of XPath query evaluation. In *PODS - Symposium on Principles of Database System*, pages 179–190. ACM Press, 2003.

[GKS04] G. Gottlob, C. Koch, and K. Schulz. Conjunctive queries over trees. In *PODS'04 :Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 189–200. ACM Press, 2004.

[GMR05] Giovanna Guerrini, Marco Mesiti, and Daniele Rossi. Impact of XML schema evolution on valid documents. In *WIDM'05 : Proceedings of the 7th annual ACM international workshop on Web information and data management*, pages 39–44, New York, NY, USA, 2005. ACM Press.

[GRS06] G. Ghelli, C. Ré, and J. Siméon. XQuery! an XML query language with side-effects. In *DATA-X colocated with EDBT 2006*, 2006.

[GRS07] Giorgio Ghelli, Kristoffer Høgsbro Rose, and Jérôme Siméon. Commutativity analysis in xml update languages. In *ICDT*, pages 374–388, 2007.

[GS97] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3. Springer Verlag, 1997.

[GT07] F. Gire and J.-M. Talbot. Visibly pushdown tree automata. In *submitted*, 2007.

[Hal96] Mírian Halfeld Ferrari Alves. *Règles pour les mises à jour des bases de données déductives*. PhD thesis, LRI, Université de Paris-Sud, 1996.

[HB03] Rachid Hamadi and Boualem Benatallah. A Petri net-based model for web service composition. In Klaus-Dieter Schewe and Xiaofang Zhou, editors, *Fourteenth Australasian Database Conference (ADC2003)*, volume 17 of *CRPIT*, pages 191–200, Adelaide, Australia, 2003. ACS.

[HBCS03] Richard Hull, Michael Benedikt, Vassilis Christophides, and Jianwen Su. E-services : a look behind the curtain. In *PODS - Symposium on Principles of Database System*, pages 1–14. ACM Press, 2003.

[HLS98] Mirian Halfeld Ferrari Alves, Dominique Laurent, and Nicolas Spyratos. Update rules in datalog programs. *J. Log. Comput.*, 8(6) :745–775, 1998.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory Languages and Computation*. Addison-Wesley Publishing Company, second edition, 2001.

[HR95] Hendrik Jan Hoogeboom and Grzegorz Rozenberg. Dependence graphs. In Volker Diekert and Grzegorz Rozenberg, editors, *The Book of Traces*. World Scientific, 1995.

[HT06] Sven Hartmann and Thu Trinh. Axiomatising functional dependencies for XML with frequencies. In *FoIKS*, pages 159–178, 2006.

[IB02] Intalio and BPMI.org. Bussiness process modeling language. Available at http ://www.bpmi.org/bpmi-downloads/BPML-SPEC-1.0.zip, 2002.

[JNV04] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. Dtds versus xml schema : A practical study. In *WebDB*, pages 79–84, 2004.

[KLP02] Eila Kuikka, Paula Leinonen, and Martti Penttonen. Towards automating of document structure transformations. In *DocEng'02 : Proceedings of the 2002 ACM symposium on Document engineering*, pages 103–110, New York, NY, USA, 2002. ACM Press.

[KLV00] George Karakostas, Richard J. Lipton, and Anastasios Viglas. On the complexity of intersecting finite state automata. In *IEEE Conference on Computational Complexity*, 2000.

[KP06] Raman Kazhamiakin and Marco Pistore. Static verification of control and data inweb service compositions. In *ICWS*, pages 83–90, 2006.

[KSS03] Nils Kalrlund, Thomas Schwentick, and Dan Suciu. XML : Model, schemas, types, logics and queries. XML Research survey (draft version), revised version to appear in Logics of emerging applications of databases, Eds. J. Chomicki, G. Saake, and R. van der Meyden, Springer Verlag, 2003.

[Ley01] F. Leyman. Web services flow language (wsfl) 1.0. Availabre at http ://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf/, 2001.

[Lib04] L. Libkin. *Elements of Finite Model Theory*. Springer-Verlag, 2004.

[Lib06] L. Libkin. Logics over unranked trees : an overview. *Logical Methods in Computer Science*, 2, 2006.

[Lim07] Maria-Adriana Lima. *Contraites d'intégrité en XML*. PhD thesis, LI, Université François Rabelais de Tours (In preparation), 2007.

[LL99] M. Levene and G. Loizou. *A guided tour of relational databases and beyond*. Springer-Verlag, 1999.

[LLL02] Mong-Li Lee, Tok Wang Ling, and Wai Lup Low. Designing functional dependencies for XML. In *Extending Database Technology (EDBT)*, pages 124–141, 2002.

[LLSV01] D. Laurent, J. Lechtenborger, N. Spyratos, and G. Vossen. Monotonic complements for independent data warehouses. *The VLDB Journal*, 10(4), 2001.

[LMM00] D. Lee, M. Mani, and M. Murata. Reasoning about XML schema languages using formal language theory. Technical report, IBM Almaden Research, 2000.

[LS06] S. Limet and G. Salzer. Tree tuple languages from the logic programming point of view. *Journal of Automated Reasoning*, 37(4) :323–349, November 2006.

[LVL03] J. Liu, M. W. Vincent, and C. Liu. Functional dependencies, from relational to XML. In *Ershov Memorial Conference*, pages 531–538, 2003.

[Mar04] Maarten Marx. Conditional XPath, the first order complete xpath dialect. In *PODS - Symposium on Principles of Database System*, pages 13–22, New York, NY, USA, 2004. ACM Press.

[Maz87] Antoni Mazurkiewicz. Trace theory. Number 255 in Lecture Notes in Computer Science (LNCS). Springer-Verlag, 1987.

[Maz95] Antoni Mazurkiewicz. Introduction to trace theory. In Volker Diekert and Grzegorz Rozenberg, editors, *The Book of Traces*. World Scientific, 1995.

[MB02] W. S. Means and M. A. Bodie. *The Book of SAX : The Simple API for XML*. No Starch Press, 2002.

[MBSR06] Tarek Melliti, Celine Boutrous-Saab, and Sylvain Rampacek. Verifying correctness of web services choreography. In *ECOWS*, pages 306–318, 2006.

[MdR05] Maarten Marx and Maarten de Rijke. Semantic characterizations of navigational XPath. *SIGMOD Rec.*, 34(2) :41–46, 2005.

[MLM01] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema language using formal language theory. In *Extreme Markup Language, Montreal, Canada*, 2001.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Inter. Tech.*, 5(4) :660–704, 2005.

[MM04] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6) :51–59, 2004.

[MP06] M. Musicante and E. Potrich. Expressing workflow patterns for web services : The case of PEWS. *Journal of Universal Computer Science*, 12(9), september 2006.

[Nev99] F. Neven. *Design and Analysis of Query Languages for Structured Documents*. PhD thesis, U. Limburg, 1999.

[Nev02] F. Neven. Automata, logic and XML. In *CSL02 - Annual Conference of the European Association for Computer Science Logic (invited talk). Available in* `http ://alpha.uhasselt.be/ fneven/publs.html`, 2002.

[Ofl96] K. Oflazer. Error-tolerant Finite-state Recognition with Applications to Morphological Analysis and Spelling Correction. *Computational Linguistics*, 22(1) :73–89, 1996.

[PBB+04] Marco Pistore, Fabio Barbon, Piergiorgio Bertoli, Dmitry Shaparau, and Paolo Traverso. Planning and monitoring web service composition. In *AIMSA*, pages 106–115, 2004.

[PTBM05] Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and A. Marconi. Automated synthesis of composite bpel4ws web services. In *ICWS*, pages 293–301. IEEE Computer Society, 2005.

[PV03] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *Proceedings of the International Conference on Database Theory (ICDT)*, 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.

[RCC05] P. Rety, J. Chabin, and J. Chen. R-unification thanks to synchronized-contextfree languages. In *19th Workshop on Unification (UNIF'2005)*, 2005.

[Rou03] M. de Rougemont. The correction of XML data. In *The First Franco-Japanese Workshop on Information, Search, Integration and Personalization - ISIP*, 2003.

[RS04] Mukund Raghavachari and Oded Shmueli. Efficient schema-based revalidation of xml. In *EDBT*, pages 639–657, 2004.

[SBS04] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *Proceeding of the 2nd International Conference on Web Services, IEEE*, 2004.

[SCZ04] M. Solanki, A. Cau, and H. Zedan. Augmenting semantic web service descriptions with compositional specification. *Proc. WWW, pp. 544-552. ACM*, 2004.

[Sel77] S. M. Selkow. The Tree-to-Tree Editing Problem. *Information Processing Letters*, 6(6) :184–186, 1977.

[SHS04] G. M. Sur, J. Hammer, and J. Siméon. An XQuery-based language for processing updates in XML. In *PLAN-X - Programming Language Technologies for XML A workshop colocated with POPL 2004*, 2004.

[SKR01] Hong Su, Harumi Kuno, and Elke A. Rundensteiner. Automating the transformation of XML documents. In *WIDM'01 : Proceedings of the 3rd international workshop on Web information and data management*, pages 68–75, New York, NY, USA, 2001. ACM Press.

[SM02] Larry Stockmeyer and Albert R. Meyer. Cosmological lower bound on the circuit complexity of a small problem in logic. *J. ACM*, 49(6) :753–784, 2002.

[SSM03] H. Seidl, T. Schwentick, and A. Muscholl. Numerical document queries. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, 2003.

[Tai79] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, 26(3), 1979.

[Tak75] M. Takahashi. Generalizations of regular sets and their application to a study of context-free languages. *Information and Control*, 27, 1975.

[Tha67] J. W. Thatcher. Characterizing derivation trees of context free grammars through a generalization of finite automata theory. *J. Comp. System Sci.*, 1 :317–322, 1967.

[Tha01] S. Thatte. XLANG : Web services for business process design. Available at http ://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.

[Tho90] W. Thomas. Automata of infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[Tho97] W. Thomas. Languages, automata and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3. Springer Verlag, 1997.

[TIHW01] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *ACM SIGMOD*. ACM, 2001.

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Theory of Computing Systems*, 2(1) :57–81, 1968.

[Via01] V. Vianu. A web odyssey : from Codd to XML. In *PODS - Symposium on Principles of Database System*. ACM Press, 2001.

[VLL04] Millist W. Vincent, Jixue Liu, and Chengfei Liu. Strong functional dependencies and their application to normal forms in xml. *ACM Trans. Database Syst.*, 29(3) :445–462, 2004.

[VSL+04] Millist W. Vincent, Michael Schrefl, Jixue Liu, Chengfei Liu, and Solen Dogen. Generalized inclusion dependencies in XML. In *Advanced Web Technologies and Applications, 6th Asia-Pacific Web Conference (APWeb)*, pages 224–233, 2004.

[VvdA05] H. M. W. Verbeek and W. M. P. van der Aalst. Analyzing BPEL processes using Petri nets. In D. Marinescu, editor, *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78, Miami, Florida, USA, 2005. Florida International University.

[Win90] M. Winslett. *Updating Logical Databases*. Cambridge University Press, 1990.

[WT05] Junhu Wang and Rodney Topor. Removing XML data redundancies using functional and equality-generating dependencies. In *ADC '05 : Proceedings of the 16th Australasian database conference*, pages 65–74, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[Xer] The Apache XML editor. Available at http ://www.apache.org/xerces-j/.

[XMLa] XML - w3c recommendation. Available at http ://www.w3.org/TR/REC-xml/.

[XMLb] XML Schema. W3C. At http ://www.w3.org/XML/Schema.

[XPaa] What's new in xpath 2.0. O Reilly XML.com At http ://www.xml.com/pub/a/2002/03/20/xpath2.html.

[XPab] XML Path Language (XPath), version 1.0. W3C. At http ://www.w3.org/TR/xpath.

[XPac] XML Path Language (XPath), version 2.0. W3C. At http ://www.w3.org/TR/xpath20/.

[XQu] XQuery 1.0 : An XML query language. W3C. At http ://www.w3.org/TR/xquery/.

[XSL] XSL Transformations (XSLT) version 1.0. W3C. At http ://www.w3.org/TR/xslt.

[Yeu06] Wing Lok Yeung. Mapping WS-CDL and BPEL into CSP for behavioural specification and verification of web services. In *ECOWS*, pages 297–305, 2006.

[ZPC97] D Ziadi, J. L. Ponty, and J.M. Champarnaud. Passage d'une expression rationnelle à un automate fini non-deterministe. *Bull. Belg. Math. Soc*, 4 :177–203, 1997.

[ZS89] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problem. *SIAM Journal on Computing*, 18(6) :1245–1262, 1989.

Annexe A

Preuve du Théorème 3.2.1

Théorème 3.2.1 : Soit L une suite non contradictoire de mises à jour élémentaires (respectant les conditions établies par la Définition 3.2.1). Soient $L_1, L_2, \dots L_m$ des suites de mises à jour construites à partir de L en changeant l'ordre des mises à jour élémentaires, c'est-à-dire, $L_1, L_2, \dots L_m$ sont des suites de mises à jour différentes entre elles uniquement par l'ordre des mises à jour élémentaires. Soit T un arbre XML et soient les mises à jour multiples (Définition 3.2.3) $T \xrightarrow{(L_1)} T'_1$, $\dots$, $T \xrightarrow{(L_m)} T'_m$.

Les arbres $T'_1 \dots T'_m$, résultats des mises à jour multiples ainsi définies, sont isomorphes. Plus particulièrement, si la liste L ne contient pas plusieurs insertions sur une même position alors les arbres résultats sont identiques (c'est-à-dire $T'_1 = \dots = T'_m = T'$). □

Preuve : La preuve est faite par induction sur la longueur de L.

Base : Si $|L| = 1$ alors nous sommes dans le cas trivial des mises à jour élémentaires. Nous considérons donc le cas de $|L| = 2$. Soient $upd_1 = (op_1, p_1, \Gamma_1)$ et $upd_2 = (op_2, p_2, \Gamma_2)$ les deux mises à jour dans L. Nous analysons tous les cas possibles.

NOTATION : Sur un arbre $T = (D, t)$ (éventuellement $T_i = (D_i, t_i)$, avec un indice) la fonction *root* est appliquée sur t pour indiquer le label de racine, c'est-à-dire, $root(t) = t(\epsilon)$.

1. $op_1 = op_2 = insert$
   (a) Soit $p_1 = p_2 = p$, c'est-à-dire, deux opérations d' insertion sur une même position.
   Supposons d'abord que upd_1 est la première mise à jour de la liste L. Soit $p = u.i$. La mise à jour upd_1 provoque un *ShiftRight* d'une position sur l'arbre original $T = T_0$ et sur le corps de la liste L. Ainsi, après que upd_1 est considérée, la liste de mises à jour est $L_1 = [(insert, u.(i+1), \Gamma_2)]$. Si $u.i \in D$, alors la deuxième mise à jour sera faite toujours à gauche du sous arbre qui était à la position $u.i$ de l'arbre original T. Si $u.i \in fr^{ins}$, alors upd_2 sera faite à la frontière de l'arbre T_1 (obtenu en considérant la mise à jour upd_1). Les tableaux suivants résument la situation en montrant comment les frères de p sont affectés par la mise à jour multiple.
   Si $p \in D$: Supposons que T, à la position p, a une étiquette A.

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$	$t_1(u.i) = root(\tau_1)$ $t_1(u.(i+1)) = A$	$t_2(u.i) = root(\tau_1)$ $t_2(u.(i+1)) = root(\tau_2)$ $t_2(u.(i+2)) = A$

   Si $p \in fr^{ins}$:

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
Position $u.i$ n'est pas dans D	$t_1(u.i) = root(\tau_1)$	$t_2(u.i) = root(\tau_1)$ $t_2(u.(i+1)) = root(\tau_2)$

   Supposons maintenant que upd_2 est la première mise à jour de la liste L. Les tableaux suivant résument la situation
   Si $p \in D$: Supposons que T, à la position p, a une étiquette A.

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$	$t_1(u.i) = root(\tau_2)$ $t_1(u.(i+1)) = A$	$t_2(u.i) = root(\tau_2)$ $t_2(u.(i+1)) = root(\tau_1)$ $t_2(u.(i+2)) = A$

Si $p \in fr^{ins}$:

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
Position $u.i$ n'est pas dans D	$t_1(u.i) = root(\tau_2)$	$t_2(u.i) = root(\tau_2)$ $t_2(u.(i+1)) = root(\tau_1)$

Nous prouvons ainsi que, dans ce cas, les arbres obtenus sont **isomorphes**.

(b) Soit $p_1 \prec p_2$ (p_1 est préfixe de p_2).
Supposons $p_1 = u.i$ et $p_2 = u.i.u'$, où $u' \in \mathbb{N}^*$ et $u' \neq \epsilon$. La position p_1 est forcément dans D. En effet, si $p_1 \in fr^{ins}(T)$ (c'est-à-dire, $p_1 \notin D$) alors $p_2 \notin fr^{ins}(T)$ et $p_2 \notin D$ ce qui est contraire aux conditions de la définition 3.2.3. La position p_2 peut être dans D ou dans $fr^{ins}(T)$.
Considérons le cas où $p_2 \in D$. Soit upd_1 la première mise à jour de la liste. Cette mise à jour provoque un *ShiftRight* sur toutes les positions ayant $u.i$ comme préfixe (sur l'arbre et sur la liste L). Ainsi, l'application du *ShiftRight* sur le corps de L transforme $p_2 = u.i.u'$ en $p_2 = u.(i+1).u'$. Le tableau ci-dessous résume la situation. Supposons que la position p_1 de T est associée à une étiquette A alors que, dans l'arbre original, la position p_2 est associée à une étiquette B.

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$ $t_0(u.i.u') = B$ où $u' = w.j(w \in \mathbb{N}^*; j \in \mathbb{N})$	$t_1(u.i) = root(\tau_1)$ $t_1(u.(i+1)) = A$ $t_1(u.(i+1).u') = B$	$t_2(u.i) = root(\tau_1)$ $t_2(u.(i+1)) = A$ $t_2(u.(i+1).w.j) = root(\tau_2)$

Soit upd_2 la première mise à jour de la liste. Cette mise à jour n'a pas d'influence sur la position de upd_1 puisque $p_1 \prec p_2$. Le tableau ci-dessous résume la situation. Nous partons des mêmes suppositions que celles faites pour le cas précédant.

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$ $t_0(u.i.u') = B$ où $u' = w.j(w \in \mathbb{N}^*; j \in \mathbb{N})$	$t_1(u.i) = A$ $t_1(u.i.w.j) = root(\tau_2)$ $t_1(u.i.w.(j+1)) = B$	$t_2(u.i) = root(\tau_1)$ $t_2(u.(i+1)) = A$ $t_2(u.(i+1).w.j) = root(\tau_2)$ $t_2(u.(i+1).w.(j+1)) = B$

Considérons le cas où $p_2 \in fr^{ins}(T)$. Supposons que la position p_1 de T est associée à une étiquette A. Soit upd_1 est la première mise à jour de la liste :

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$ $t_0(u.i.u') \in fr^{ins}(T)$ où $u' = w.j(w \in \mathbb{N}^*; j \in \mathbb{N})$	$t_1(u.i) = root(\tau_1)$ $t_1(u.(i+1)) = A$ $t_0(u.(i+1).u') \in fr^{ins}(T)$	$t_2(u.i) = root(\tau_1)$ $t_2(u.(i+1)) = A$ $t_2(u.(i+1).w.j) = root(\tau_2)$ $t_2(u.(i+1).w.(j+1)) \in fr^{ins}(T)$

Soit upd_2 est la première mise à jour de la liste :

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$ $t_0(u.i.u') \in fr^{ins}(T)$ où $u' = w.j(w \in \mathbb{N}^*; j \in \mathbb{N})$	$t_1(u.i) = A$ $t_1(u.i.w.j) = root(\tau_2)$ $t_1(u.i.w.(j+1)) \in fr^{ins}(T)$	$t_2(u.i) = root(\tau_1)$ $t_2(u.(i+1)) = A$ $t_2(u.(i+1).w.j) = root(\tau_2)$ $t_2(u.(i+1).w.(j+1)) \in fr^{ins}(T)$

Nous prouvons donc que dans ce cas, les arbres obtenus sont **identiques**.

(c) Il n'y a pas une relation de descendance directe entre p_1 et p_2 et $p1$ est une position plus à gauche que $p2$.

Nous pouvons encore distinguer le cas où p_1 est un frère d'un ancêtre de p_2 ou de p_2 lui même. Dans ce contexte, supposons $p_1 = u.i$ et $p_2 = u.k.v.j$ avec $v \in \mathbb{N}^*$ et $i, k, j \in \mathbb{N}$ et $i < k$. Dans ce cas p_1 est dans D dû aux mêmes raisons que celles expliquées dans l'item 1b. Si p_2 est dans fr^{ins}, la situation est similaire.

Soit upd_1 la première insertion. Cette mise à jour provoque un *ShiftRight* sur l'arbre et la liste. La position de upd_2 change de $p_2 = u.k.v.j$ à $p_2 = u.(k+1).v.j$. Le tableau ci-dessous résume la situation. Pour faciliter la visualisation, nous considérons que dans l'arbre d'origine $t(u.i) = A$, $t(u.k) = B$ et $t(u.k.v.j) = C$.

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$ $t_0(u.k) = B$ $t_0(u.k.v.j) = C$	$t_1(u.i) = root(\tau_1)$ $t_1(u.(i+1)) = A$ $t_1(u.(k+1)) = B$ $t_1(u.(k+1).v.j) = C$	$t_2(u.i) = root(\tau_1)$ $t_2(u.(i+1)) = A$ $t_2(u.(k+1)) = B$ $t_2(u.(k+1).v.j) = root(\tau_2)$ $t_2(u.(k+1).v.(j+1)) = C$

Soit upd_2 la première mise à jour de la liste. Cette mise à jour n'a pas d'influence sur la position de upd_1 puisque p_1 est à gauche de p_2. Le tableau ci-dessous résume la situation. Nous partons des mêmes suppositions que celles faites pour le cas précédant.

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$ $t_0(u.k) = B$ $t_0(u.k.v.j) = C$	$t_1(u.i) = A$ $t_1(u.k) = B$ $t_1(u.k..v.j) = root(\tau_2)$ $t_1(u.k.v.(j+1)) = C$	$t_2(u.i) = root(\tau_1)$ $t_2(u.(i+1)) = A$ $t_2(u.(k+1)) = B$ $t_2(u.(k+1).v.j) = root(\tau_2)$ $t_2(u.(k+1).v.(j+1)) = C$

Dans ce cas, les arbres obtenus sont **identiques**.

Si p_1 et p_2 ne sont pas dans la situation considérée ci-dessus (c'est-à-dire, p_1 n'est ni un frère d'un ancêtre de p_2 ni un frère de p_2 lui même) alors, les mises à jour sont complètement indépendantes. Ainsi, upd_1 n'a pas d'influence sur la position de upd_2 et vice-versa.

Nous prouvons donc que quand aucune relation de descendance existe entre p_1 et p_2, les arbres obtenus sont **identiques**.

Ainsi, si la liste non contradictoire L d'une mise à jour multiple U contient deux opérations insert, alors, les arbres obtenus par l'application de U sont isomorphes (ou identiques, dans certains cas), indépendamment de l' ordre des opérations dans L.

2. $op_1 = delete$ et $op_2 = delete$.

D'après la définition 3.2.2, si $p_1 = p_2$ ou si $p_1 \prec p_2$ alors la mise à jour est contradictoire. Nous considérons donc seulement le cas où $p1$ est une position à gauche de $p2$ (sans lien de descendance directe). Ce cas est similaire à celui traité dans l'item 1c.

Si p_1 est un frère d'un ancêtre de p_2 ou de p_2 lui même, alors nous pouvons supposer que $p_1 = u.i$ et $p_2 = u.k.v.j$ avec $v \in \mathbb{N}^*$, $k, j \in \mathbb{N}$ et $i < k$. Si upd_1 est la première suppression, cette mise à jour provoque un *ShiftLeft* sur l'arbre et la liste. La position de upd_2 change de $p_2 = u.k.v.j$ à $p_2 = u.(k-1).v.j$. Le tableau ci-dessous résume la situation. Pour faciliter la visualisation, nous considérons que dans l'arbre d'origine $t(u.i) = A$, $t(u.(i+1)) = M$, $t(u.k) = B$ et $t(u.k.v.j) = C$.

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$ $t_0(u.k) = B$ $t_0(u.k.v.j) = C$	$t_1(u.i) = t_0(u.(i+1)) = M$ $t_1(u.(k-1)) = B$ $t_1(u.(k-1).v.j) = C$	$t_2(u.i) = M$ $t_2(u.(k-1)) = B$ $t_2(u.(k-1).v.j) = t_1(u.(k-1).v.(j+1)) = N$ si $(u.(k-1).v.(j+1))$ est dans D_1) $(*)$ sinon $u.(k-1).v.j$ n'est plus dans D_2)
	Le sous arbre dont la racine avait pour étiquette A n'existe plus.	Les sous arbres dont les racines étaient A et C n'existent plus.

$*$: Cela veut dire que le sous arbre enraciné à la position $u.(k-1).v.(j+1)$ de T_1 (s'il existe) est le même sous arbre que celui enraciné à la position $u.k.v.(j+1)$ de l'arbre original T_0 . Nous supposons que l'étiquette de cette racine (si elle existe) est N.

Soit upd_2 la première mise à jour de la liste. Cette mise à jour n'a pas d'influence sur la position de upd_1 puisque p_1 est à gauche de p_2. Le tableau ci-dessous résume la situation en partant des mêmes suppositions que ci-dessus.

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$ $t_0(u.k) = B$ $t_0(u.k.v.j) = C$	$t_1(u.i) = A$ $t_1(u.k) = B$ $t_1(u.k.v.j) = t_0(u.k.v.(j+1)) = N$; si $(u.k.v.(j+1))$ est dans D_0 $(**)$ sinon $u.k.v.j$ n'est plus dans D_1	$t_2(u.i) = T_1(u.(i+1)) = M$ $t_2(u.(k-1)) = B$ $t_2(u.(k-1).v.j) = N$; si $(u.k.v.j)$ est dans D_1
	Le sous arbre dont la racine était C n'existe plus.	Les sous arbres dont les racines étaient A et C n'existent plus.

$(**)$: Le *ShiftLeft* provoque un "mouvement" vers la gauche. Si le sous arbre à la position $u.k.v.(j+1)$ de l'arbre original T_0 existe, alors il passe à la position $u.k.v.j$ de T_1. Nous supposons que $t_0(u.k.v.(j+1)) = N$.

Cela est la preuve que dans ce cas, les arbres obtenus sont identiques.

Comme dans l'item 1c, si p_1 et p_2 ne sont pas dans la situation considérée ci-dessus, les mises à jour sont complètement indépendantes.

Nous prouvons donc que si la liste non contradictoire L d'une mise à jour multiple U contient deux opérations delete alors, les arbres obtenus par l'application de U sont ***identiques****, indépendamment de l'ordre des opérations dans L. En d'autres mots, si deux opérations delete (correctes sur un arbre T) sont non contradictoires, alors elles sont commutatives.*

3. $op_1 = replace$ et $op_2 = replace$

   D'après la définition 3.2.2, si $p_1 = p_2$ ou si $p_1 \prec p_2$ alors la mise à jour est contradictoire. Nous considérons donc seulement le cas où p_1 une position à gauche de p_2 (sans descendance directe). Néanmoins, dans ce cas, les mises à jour sont indépendantes, car un *replace* ne provoque ni un *ShiftLeft* ni un *ShiftRight* des positions frères de la position de mise à jour.

   Si la liste non contradictoire L d'une mise à jour multiple U contient deux opérations *replace*, alors, les arbres obtenus par l'application de U sont **identiques**, indépendamment de l' ordre des opérations dans L. En d'autres mots, *si deux opérations replace (correctes sur un arbre T) sont non contradictoires, alors elles sont commutatives.*

4. $op_1 = insert$ et $op_2 = delete$

   (a) Si $p_1 = p_2 = p$, alors la mise à jour correspond à un *replace* (on suppose que le pré-traitement fait le remplacement).

   (b) Soit $p_1 \prec p_2$ (p_1 est préfixe de p_2). Remarquer que si $p_2 \prec p_1$ alors la liste de mise à jour est contradictoire.

   La preuve est similaire à celle faite dans l'item 1b. Supposons $p_1 = u.i$ est $p_2 = u.i.u'$, où $u' \in \mathbb{N}^*$ et $u' \neq \epsilon$. La position p_2 est forcément dans D (définition 3.2.3) et donc p_1 aussi.

   Soit upd_1 la première mise à jour de la liste. Cette mise à jour provoque un *ShiftRight* sur toutes les positions ayant $u.i$ comme préfixe (sur l'arbre et sur la liste L). Ainsi, l'application du *ShiftRight* sur le corps de L transforme $p_2 = u.i.u'$ en $p_2 = u.(i+1).u'$. Le tableau ci-dessous résume la situation. Supposons que la racine du sous arbre de T à la position p_1 a une étiquette A alors que la racine du sous arbre de T à la position p_2 a une étiquette B .

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$ $t_0(u.i.w.j) = B$ $t_0(u.i.w.(j+1)) = C$ si $u.i.w.(j+1)$ est dans D_0	$t_1(u.i) = root(\tau_1)$ $t_1(u.(i+1)) = A$ $t_1(u.(i+1).w.j) = B$ $t_1(u.(i+1).w.(j+1)) = C$	$t_2(u.i) = root(\tau_1)$ $t_2(u.(i+1)) = A$ $t_2(u.(i+1).w.j) = C$
		Le sous arbre dont la racine était B n'existe plus

Soit upd_2 la première mise à jour de la liste. Cette mise à jour n'a pas d'influence sur la position de upd_1 puisque $p_1 \prec p_2$. Le tableau ci-dessous résume la situation. Nous partons des mêmes suppositions faites pour le cas précédent.

Arbre original	Arbre après la première mise à jour	Arbre après la deuxième mise à jour
$t_0(u.i) = A$ $t_0(u.i.w.j) = B$ $t_0(u.i.w.(j+1)) = C$ si $u.i.w.(j+1)$ est dans D_0	$t_1(u.i) = A$ $t_1(u.i.w.j) = C$	$t_2(u.i) = root(\tau_1)$ $t_2(u.(i+1)) = A$ $t_2(u.(i+1).w.j) = C$ si $u.i.w.(j+1)$ est dans D_0
	Le sous arbre dont la racine était B n'existe plus	

Nous prouvons donc que dans ce cas, les arbres obtenus sont **identiques**.

(c) Considérons le cas où il n'y a pas une relation de descendance entre p_1 et p_2 mais une position est à gauche de l'autre. Dans ce cas, de manière similaire à l'item 1c, nous devons considérer le cas où p_1 et p_2 sont frères ou l'un est frère d'un ancêtre de l'autre.

Supposons d'abord que $p_1 = u.i$ et $p_2 = u.k.v.j$ avec $v \in \mathbb{N}^*$ et $k, j \in \mathbb{N}$ $(i < k)$. Dans ce cas les deux positions sont dans D. Soit $t(u.k.v.j) = C$. L'insertion upd_1 provoque un *ShiftRight* sur l'arbre et la liste. La position de upd_2 change de $p_2 = u.k.v.j$ à $p_2 = u.(k+1).v.j$. Comme le changement est le même sur la liste de mise à jour et sur l'arbre, upd_2 supprime le sous arbre dont la racine est C. Si la mise à jour upd_2 est la première de la liste, alors la suppression upd_2 élimine le sous arbre dont la racine est C. La position p_1 ne change pas puisque p_1 est à gauche de p_2. Ensuite, upd_1 ajoute un nouveau sous arbre à la position $p_1 = u.i$ provoquant un *ShiftRight* sur l'arbre. L'arbre mis à jour via la suite $(upd_2 ; upd_1)$ est donc **identique** à celui obtenu par la suite $(upd_1 ; upd_2)$.

Supposons maintenant le contraire, c'est-à-dire, $p_2 = u.i$ et $p_1 = u.k.v.j$ $(i < k)$. Dans ce cas p_2 est dans D mais p_1 peut être à la frontière d'insertion.

Supposons que upd_1 est la première mise à jour. Si $t(u.k.v.j) = E$, alors l'insertion upd_1 ajoute un nouveau sous arbre comme frère gauche de celui dont la racine est E. Si $(u.k.v.j) \in fr^{ins}(T)$ alors upd_1 ajoute un nouveau sous arbre à une extrémité de T. La position de upd_2 ne change pas, puisque p_2 est à gauche de p_1. La suppression upd_2 provoque un *ShiftLeft* sur l'arbre.

Supposons que upd_2 est la première mise à jour. La suppression upd_2 provoque un *ShiftLeft* sur l'arbre et la liste. La position de upd_1 change de $p_1 = u.k.v.j$ à $p_1 = u.(k-1).v.j$. Comme le changement est le même sur la liste de mise à jour et sur l'arbre, si originalement $t(u.k.v.j) = E$, alors $t_1(u.(k-1).v.j) = E$ et upd_1 ajoute un nouveau sous arbre comme frère gauche de celui dont la racine est E. Si dans l'arbre d'origine $(u.k.v.j) \in fr^{ins}(T)$, alors $u.(k-1).v.j \in fr^{ins}(T_1)$ et la deuxième mise à jour ajoute un nouveau sous arbre à une extrémité de T_1.

*Si la liste non contradictoire L d'une mise à jour multiple U contient les opérations (correctes sur T) delete et insert, alors, les arbres obtenus par l'application de U sont **identiques**, indépendamment de l'ordre des opérations dans L.*

5. $op_1 = insert$ et $op_2 = replace$

   À partir des preuves précédentes, il est facile de prouver que si la liste non contradictoire L d'une mise à jour multiple U contient les opérations (correctes sur T) *insert* et *replace*, alors, les arbres obtenus par l'application de U sont **identiques**, indépendamment de l'ordre des opérations dans L. Cela est dû aux faits suivants : (a) l'opération *replace* ne provoque pas de *shift* et (b) le *ShiftRight* provoqué par l'opération *insert* est appliqué non seulement à l'arbre mais aussi aux mises à jour encore dans L dont les positions sont des descendants de p_1 ou des positions à droite de p_1.

6. $op_1 = delete$ et $op_2 = replace$

   La seule possibilité de ne pas avoir des opérations contradictoires coïncide avec le cas où il n'y a pas une relation de descendance directe entre p_1 et p_2 mais une position est à gauche de l'autre. Dans ce cas, de manière similaire à ce que nous avons fait précédemment, nous devons considérer le cas où p_1 et p_2 sont frères ou l'un est frère d'un ancêtre de l'autre. La preuve est simple et similaire aux précédentes.

Ainsi, si la liste non contradictoire L d'une mise à jour multiple U contient les opérations (correctes sur T) *delete* et *replace*, alors, les arbres obtenus par l'application de U sont identiques, indépendamment de l'ordre des opérations dans L.

Induction : Nous supposons $|L| \geq 2$. Soit n un entier arbitraire et supposons que L_n est une liste avec n mises à jour interchangeables. Nous avons donc $n!$ façon d'écrire cette liste.

Soit L la liste avec $n+1$ mises à jour obtenue par l'ajout d'une nouvelle mise à jour upd_{n+1} à L_n. Soit i la position de la mise à jour upd_{n+1} dans L. Nous pouvons échanger cette mise à jour avec n'importe quelle mise à jour upd dans une position $j \neq i$ de L :

- Si $i < j$, échanger upd avec la mise à jour dans la position $j-1$ de L. Continuer les échanges jusqu'à placer la mise à jour upd à la position $i+1$. Tous ces changements sont possibles car, par l'hypothèse d'induction les mises à jours de L_n sont interchangeables.
- Échanger la mise à jour upd_{n+1} (position i) avec upd (position $i+1$). Cela est possible, selon la preuve de la base de l'induction.
- Pour $i > j$ le raisonnement est symétrique.

Nous avons donc $(n+1)!$ façon d'écrire cette liste. □

Annexe B

Les étapes de la transformation des expressions régulières

Dans les sections précédentes les idées générales du calcul des expressions régulières dans `GREC` et `dGREC` ont été présentées. Dans `dGREC`, l'expression régulière originale est décomposée en sous-expressions et le calcul est fait pour chaque sous-expression. Dans `GREC`, la hiérarchie des orbites du graphe de Glushkov guide la procédure de réduction et transformation. Dans cette section nous considérons les modifications faites sur l'expression régulière originale. Nous montrons les conditions testées et les changements effectués en faisant un parallèle entre les versions `dGREC` et `GREC`. Le but est de montrer que dans les deux approches les mêmes cas sont considérés et les mêmes solutions sont proposées.

RAPPEL : Dans les deux versions, remarquer que, pour simplifier la notation, nous utilisons le symbole ! pour représenter ? ou *. Ainsi, l'expression $a.n!$ est un raccourci pour $a.n?$ ou $a.n^*$.

Dans l'algorithme de la figure 5.4, la fonction `LookForGrapheAlternative` a un rôle essentiel puisqu'elle est la responsable de la construction de nouveaux graphes (correspondant aux expressions régulières candidates). `LookForGrapheAlternative` utilise les règles $\mathbf{R}_1$, $\mathbf{R}_2$, $\mathbf{R}_3$, ainsi qu'une règle $\mathbf{R}_4$ introduite pour régler la situation des fermetures de Kleene, pour guider les modifications apportées au graphe à réduire. Nous ne présentons pas dans ce mémoire les définitions complètes pour chaque modification (elles sont présentées dans [BDH$^+$04b, Dua05]). Les figures B.1 à B.3 montrent les testes (colonne **Condition**) et les modifications effectuées quand les conditions testées sont satisfaites (colonne **Result**).

Ce mémoire présente une version révisée de celle présentée dans [Dua05]. En effet, dans [Dua05] étant donné un nœud x du graphe G_{wo}, les ensembles $Q^+(x)$ et $Q^-(x)$ sont utilisés dans les conditions à tester, pour déterminer les modifications que `GREC` doit effectuer. Dans la version révisée que nous proposons ici, la procédure de réduction reste la même (c'est-à-dire, celle introduite dans [CZ00] et qu'utilise $Q^+(x)$ et $Q^-(x)$), mais les conditions à tester pour définir les solutions de `GREC` ne sont plus construites sur $Q^+(x)$ et $Q^-(x)$. En effet, ces conditions sont construites sur les ensembles $Foll(x)$ et $Prev(x)$ définis dans la suite :

Définition B.0.1 - Ensemble des nœuds prédécesseurs et des nœuds successeurs dans un graphe : Étant donné un un graphe $G = (X, U)$, et un nœud $x \in X$, définir $Foll(x) = \{y \in X \mid (x, y) \in U\}$ et $Prev(x) = \{y \in X \mid (y, x) \in U\}$. □

Remarquer que la différence entre les couples d'ensemble $Foll(x)$ et $Prev(x)$ par rapport à $Q^+(x)$ et $Q^-(x)$ (définition 5.2.1) est le graphe d'origine de leurs définitions. Les premiers sont définis sur un graphe de Glushkov alors que les deuxièmes sont définis sur le graphe sans orbite.

Dans la section 5.3 le calcul de `dGREC` est fondé sur le résultat des fonctions *evAnd*, *evOr*, *evOpt* et *evClosure*. Ces fonctions effectuent des testes qui sont construits avec les fonctions présentées dans la section 5.1.2. De plus, elles utilisent la fonction $Update(E, F, G)$ qui retourne une nouvelle expression régulière E', obtenue en remplaçant la sous-expression F par la sous-expression G dans E. Par exemple, si $E = a_1.b_2.(c_3 + d_4)$, $F = (c_3 + d_4)$ et $G = g_5?.(c_3 + d_4)$, alors $Update(E, F, G)$ rend comme résultat

$E' = a_1.b_2.g_5?.(c_3 + d_4)$.

Nous considérons que x et y sont deux nœuds du graphe G_{wo} sur lesquels les règles de réduction peuvent être appliquées. Nous remarquerons donc, dans la suite, que les cas testés et les solutions proposées sont les mêmes décrits pour les deux versions. Nous comparons les fonctions *evAnd*, *evOr*, *evOpt* et *evClosure* aux conditions et transformations proposées par `GREC`, selon les règles $\mathbf{R}_1$, $\mathbf{R}_2$, $\mathbf{R}_3$ et $\mathbf{R}_4$.

B.0.1 Changement dans une sous-expression AND

Il s'agit de l'insertion dans une sous-expression de la forme $F.G$. Deux cas sont considérés. Dans le premier cas, nous testons si l'insertion est faite entre un dernier symbole de F et un premier symbole de G. La solution consiste à concaténer le nouveau symbole à la fin de F. Par exemple, supposons que $E = a_1.b_2$ et $w = ab$. Si $w' = anb$, alors $s_{nl} = 1$ et $s_{nr} = 2$ et les expressions régulières candidates sont $a_1.n_3?.b_2$ et $a_1.n_3^*.b_2$. Le deuxième cas arrive quand la sous-expression $F.G$ peut se répéter. Le teste vérifie si un nouveau symbole apparaît entre un dernier symbole de G et un premier symbole de F. La solution consiste à concaténer le nouveau symbole à la fin de G ou au début de F. Par exemple, si $E = a_1.(b_2.c_3)^*.d_4$, $w = abcbcd$ et $w' = abcnbcd$, avec $s_{nl} = 3$ et $s_{nr} = 2$. Les nouvelles expressions régulières sont $E = a_1.(b_2.c_3.n_5!)^*.d_4$ et $E = a_1.(n_5!.b_2.c_3)^*.d_4$.

A- Dans `dGREC` : $evAnd(F, G, E, s_{nl}, s_{nr}, s_{new})$ est l'ensemble d'expressions régulières obtenu par l'évolution de la sous-expression $F.G$ dans l'expression originale E. $evAnd(F, G, E, s_{nl}, s_{nr}, s_{new})$ est calculé par Algorithme 4 :

Algorithm 4 $evAnd(F, G, E, s_{nl}, s_{nr}, s_{new})$

```
var R := ∅                                                          // Solutions calculated so far.
if s_nl ∈ Last(F) ∧ s_nr ∈ First(G) then
   R := R ∪ Update(E, F, F.s_new!)                                  // Case 1.
end if
if s_nr ∈ First(F) ∧ s_nl ∈ Last(G) then
   R := R ∪ Update(E, F, s_new!.F) ∪ Update(E, G, G.s_new!)         // Case 2.
end if
return R
```

B- Dans `GREC` : C'est la règle $\mathbf{R}_1$ qui définit cette situation. La figure B.1 montre comment `LookForGraphAlternative` construit des nouveaux graphes quand la règle $\mathbf{R}_1$ est appliquée.

Soient x et y deux nœuds du graphe G_{wo} sur lesquels R_1 peut être appliquée. Il s'agit de tester, pour le premier cas, si (i) le nœud x correspond à s_{nl} et (ii) le nœud y correspond à s_{nr}. Le deuxième cas arrive quand s_{nl} est une porte de sortie d'une orbite $\mathcal{O}$ et si s_{nr} est une porte d'entrée de $\mathcal{O}$.

B.0.2 Changement dans une sous-expression OR

Il s'agit de l'insertion dans une sous-expression de la forme $F + G$. Si l'insertion se faite entre un dernier symbole de F et un successeur de ce symbole, la solution consiste à concaténer le symbole inséré à la fin de F. Par exemple, si $E = a_1.(b_2 + c_3).d_4$, $w = abd$ et $w' = abnd$, avec $s_{nl} = 2$ et $s_{nr} = 4$ alors l'expression candidate est $E' = a_1.(b_2.n_5! + c_3).d_4$. Pour une situation symétrique, une solution symétrique est proposée. Si l'insertion se faite entre un prédécesseur d'un premier symbole de F et le successeur d'un dernier symbole de F, la solution consiste à créer une nouvelle branche du OR. Par exemple, si $E = a_1.(b_2 + c_3)?.d_4$, $w = ad$ et $w' = and$, alors $E' = a_1.(b_2 + c_3 + n_5!)?.d_4$.

A- Dans `dGREC` : $evOr(F, G, E, s_{nl}, s_{nr}, s_{new})$ est l'ensemble d'expressions régulières obtenu par l'évolution de la sous-expression $F + G$ dans l'expression originale E. $evOr(F, G, E, s_{nl}, s_{nr}, s_{new})$ est calculé par Algorithme 5.

B- Dans `GREC` : C'est la règle $\mathbf{R}_2$ qui définit cette situation. La figure B.2 montre comment `LookForGraphAlternative` construit des nouveaux graphes quand la règle $\mathbf{R}_2$ est appliquée.

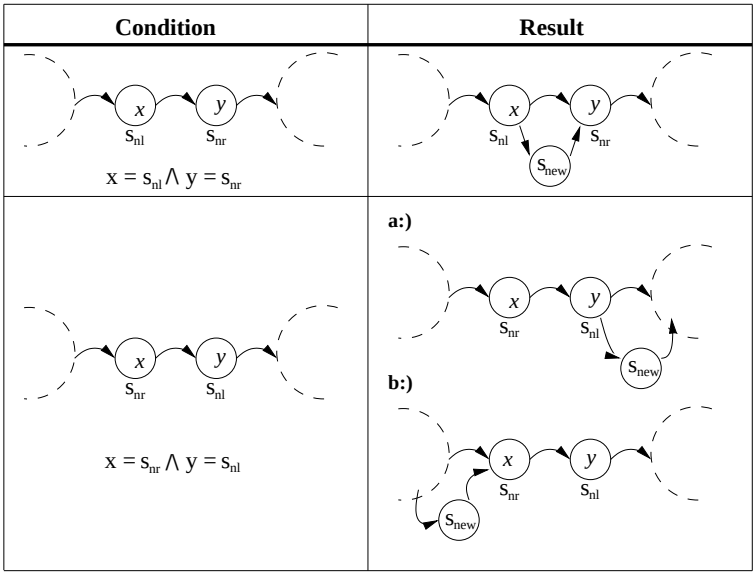


FIG. B.1: Conditions et modifications à tester avant d'appliquer la règle $\mathbf{R}_1$.

Algorithm 5 $evOr(F, G, E, s_{nl}, s_{nr}, s_{new})$

```
var R := ∅                                                    // Solutions calculated so far.
for (X := F, G) do
   if s_nl ∈ Last(X) ∧ s_nr ∈ Follow(E, s_nl) then
      R := R ∪ Update(E, X, X.s_new!)                         // Case 1.
   end if
end for
for (X := F, G) do
   if s_nr ∈ First(X) ∧ s_nl ∈ Previous(E, s_nr) then
      R := R ∪ Update(E, X, s_new!.X)                         // Case 2.
   end if
end for
if s_nl ∈ Previous(E, First(F)) ∧ s_nr ∈ Follow(E, Last(F)) then
   R := R ∪ Update(E, G, G + s_new!)                          // Case 3.
end if
return R
```

B.0.3 Changement dans une sous-expression optionnelle

Il s'agit de l'insertion dans une sous-expression de la forme $F?$. Si le nouveau symbole est inséré entre un prédécesseur d'un premier symbole de F et un successeur d'un dernier symbole de F, alors trois différentes solutions peuvent être proposées. Par exemple, si $E = a_1.b_2?.c_3$, $w = ac$ et $w' = anc$, avec $s_{nl} = 1$ et $s_{nr} = 3$, les nouvelles expressions régulières proposées sont $E'_1 = a_1.(n_4!.b_2)?.c_3$, $E'_2 = a_1.(b_2.n_4!)?.c_3$ et $E'_3 = a_1.(n_4! + b_2)?.c_3$.

A- Dans `dGREC` : $evOpt(F, E, s_{nl}, s_{nr}, s_{new})$ est l'ensemble d'expressions régulières obtenu par l'évolution de la sous-expression $F?$ dans l'expression originale E. $evOpt(E, F, s_{nl}, s_{nr}, s_{new})$ est calculé par Algorithme 6.

Algorithm 6 $evOpt(F, E, s_{nl}, s_{nr}, s_{new})$

```
var R := ∅                                                    // Solutions calculated so far.
if s_nl ∈ Previous(E, First(F)) ∧ s_nr ∈ Follow(E, Last(F)) then
   R := R ∪ Update(E, F, s_new!.F) ∪ Update(E, F, F.s_new!) ∪ Update(E, F, s_new! + F)
end if
return R
```

B- Dans `GREC` : C'est la règle $\mathbf{R}_3$ qui définit cette situation. La figure B.3 montre comment `LookForGraphAlternative` construit des nouveaux graphes quand la règle $\mathbf{R}_3$ est appliquée.

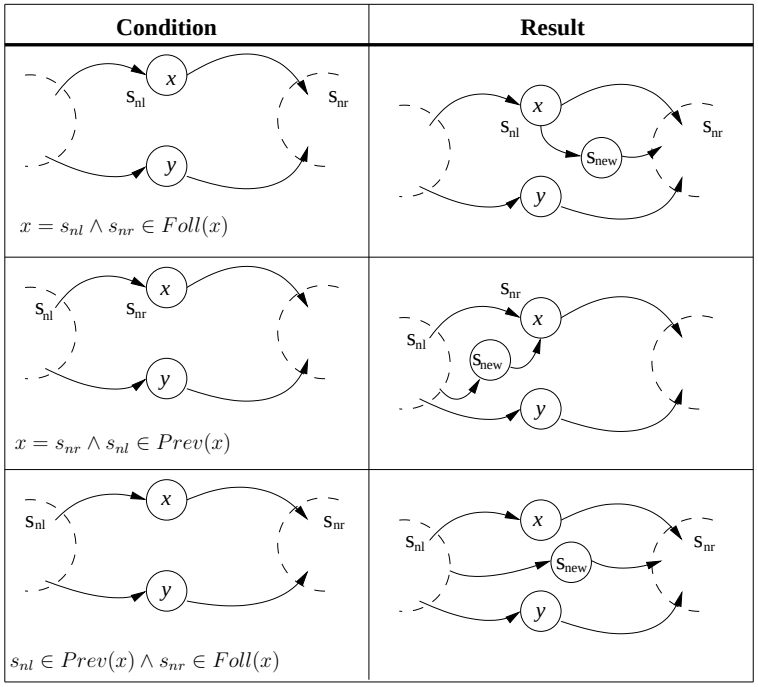


FIG. B.2: Conditions et modifications à tester avant d'appliquer la règle $\mathbf{R}_2$.

B.0.4 Changement dans une sous-expression étoilée

Il s'agit de l'insertion dans une sous-expression de la forme F^* ou F^+. Si le nouveau symbole est inséré entre un dernier et un premier symbole de F deux différentes solutions sont proposées. Par exemple, si $E = a_1^*$, $w = aa$ et $w' = ana$, où $s_{nl} = 1$ et $s_{nr} = 1$, nous obtenons $E'_1 = (n_2!.a_1)^*$ et $E'_2 = (a_1.n_2!)^*$. Si le nouveau symbole est inséré entre un prédécesseur d'un premier symbole de F et un premier symbole de F, alors deux solutions sont proposées. Par exemple, $E = a_1.b_2^*$, $w = abb$ et $w' = anbb$, nous avons $s_{nl} = 1$ et $s_{nr} = 2$ et $E'_1 = a_1.(n_3!.b_2)^*$ et $E'_2 = a_1.(n_3 + b_2)^*$. Le troisième cas est symétrique au deuxième.

A- Dans `dGREC` : $evClosure(F, E, s_{nl}, s_{nr}, s_{new})$ est l'ensemble d'expressions régulières obtenu par l'évolution de la sous-expression F^+ ou F^* dans l'expression originale E. $evClosure(E, F, s_{nl}, s_{nr}, s_{new})$ est calculé par Algorithme 7.

Algorithm 7 $evClosure(F, E, s_{nl}, s_{nr}, s_{new})$

var $R := \emptyset$ // Solutions calculated so far.
if $s_{nl} \in Last(F) \ \wedge \ s_{nr} \in First(F)$ **then**
  $R := R \ \cup \ Update(E, F, s_{new}!.F) \ \cup \ Update(E, F, F.s_{new}!) \cup \ Update(E, F, F + s_{new}!)$ // Case 1.
end if
if $s_{nr} \in First(F) \wedge \ \ s_{nl} \in Previous(E, s_{nr})$ **then**
  $R := R \ \cup \ Update(E, F, s_{new}!.F) \ \cup \ Update(E, F, s_{new} + F)$ // Case 2.
end if
if $s_{nl} \in First(F) \ \wedge \ s_{nr} \in Follow(E, s_{nl})$ **then**
  $R := R \ \cup \ Update(E, F, F.s_{new}!) \ \cup \ Update(E, F, s_{new} + F)$ // Case 3.
end if
return R

B- Dans `GREC` : 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 noeud, qui contient l'expression qui sera décorée avec un plus ($^+$). Avant d'appliquer la décoration, on doit considérer l'insertion du noeud s_{new} dans l'orbite $\mathcal{O}$. La figure B.4 résume les conditions dans lesquelles le graphe d'origine peut être changé à cette étape de la réduction.

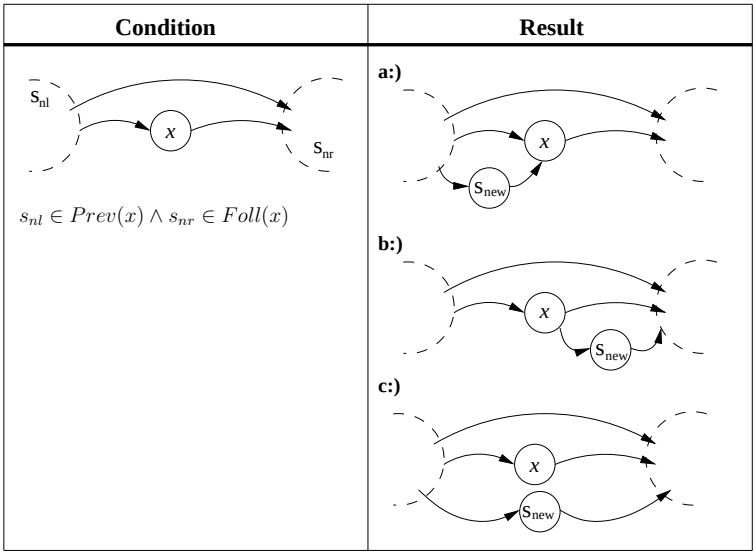


FIG. B.3: Conditions et modifications à tester avant d'appliquer la règle $\mathbf{R}_3$.

B.0.5 Les suppressions

Les suppressions sont plus simples que les insertions : une suppression correspond à rendre un symbole optionnel. Dans `GREC` cela signifie qu'il faut transformer un état de l'automate M_E en état optionnel (figure B.5). La procédure `LookForGraphAlternative` est appelée en sachant que maintenant s_{new} représente un nœud existant dans le graphe. 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 noeuds déjà optionnels.

Dans l'approche `dGREC`, l'opération $Delete(E, p)$ est introduite.

Définition B.0.2 $Delete(E, p)$ est l'expression régulière obtenue par l'évolution d'une expression régulière E, en rendant optionnelle la position p de E. La fonction $Delete(E, p)$ peut être définie inductivement sur la structure de l'expression régulière E.

$Delete(E, p)$ peut être inductivement calculée de la façon suivante :

$$
\begin{aligned}
Delete(\emptyset, p) &= \emptyset \\
Delete(\varepsilon, p) &= \varepsilon \\
Delete(\alpha_x, p) &= \begin{cases} \alpha_x? & \text{if } x = p \\ \alpha_x & \text{otherwise} \end{cases} \\
Delete(F + G, p) &= \begin{cases} Delete(F, p) + G & \text{if } p \in Pos(F) \\ F + Delete(G, p) & \text{if } p \in Pos(G) \end{cases} \\
Delete(F.G, p) &= \begin{cases} Delete(F, p).G & \text{if } p \in Pos(F) \\ F.Delete(G, p) & \text{if } p \in Pos(G) \end{cases} \\
Delete(F^+, p) &= \begin{cases} F^* & \text{if } |Pos(F)| = 1 \wedge Pos(F) = p \\ (Delete(F, p))^+ & \text{otherwise} \end{cases} \\
Delete(F^*, p) &= (Delete(F, p))^* \\
Delete((F), p) &= (Delete(F, p)) \\
Delete(F?, p) &= (Delete(F, p))?
\end{aligned}
$$

□

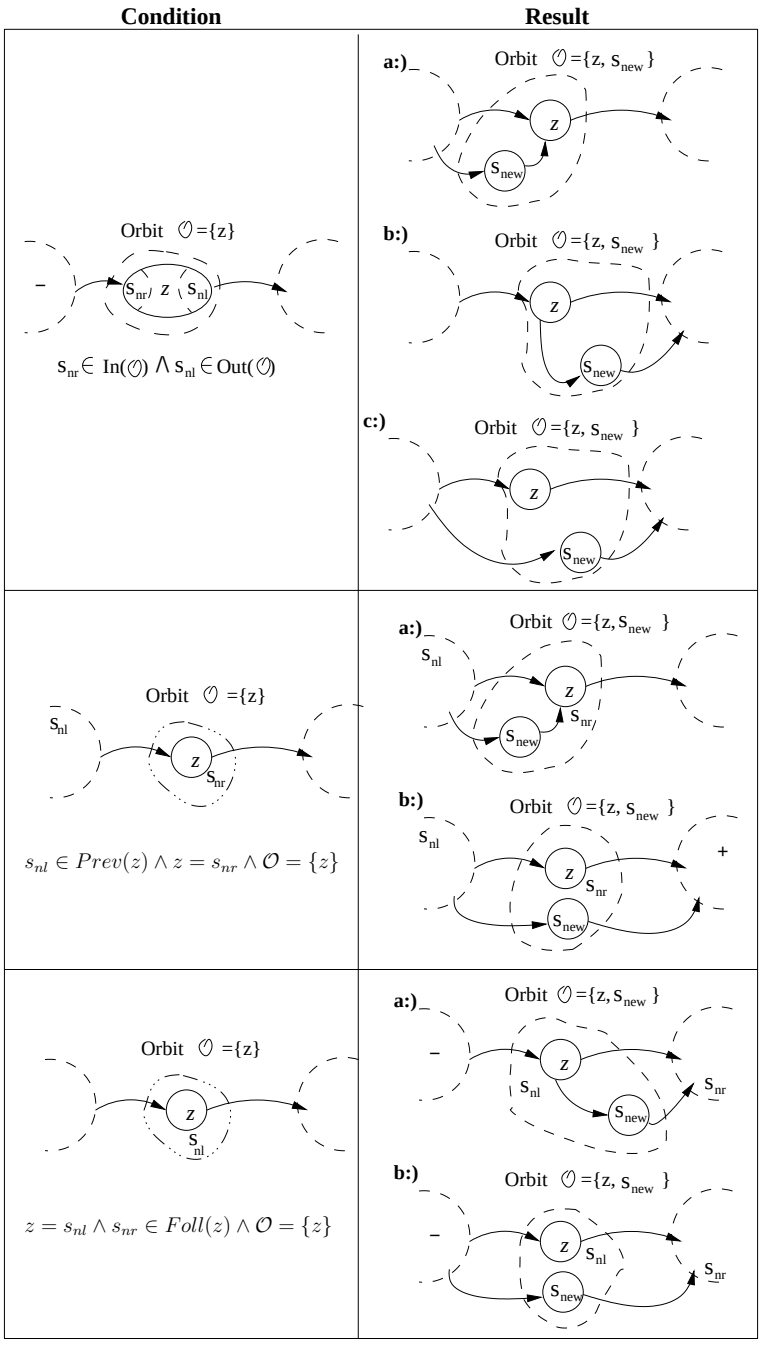


FIG. B.4: Conditions et modifications à tester après la réduction d'une orbite.

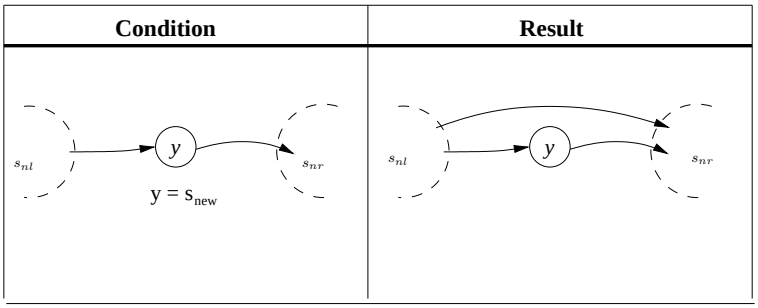


FIG. B.5: Conditions et modifications pour transformer un nœud obligatoire en optionnel

Annexe C

Publications représentatives

L'annexe contient des publications représentatives des sujets traités dans ce mémoire de HDR. La plupart de mes publications sont disponibles sur ma page web[1] ou peuvent être demandées via email.

Dans ce mémoire j'ai décrit mes travaux dans le domaine de XML en les complétant, quelques fois, par des précisions qui ne sont pas dans les publications (par exemple, la commutativité des mises à jour non contradictoires, le parallèle entre `dGREC` et `GREC` ainsi que l'utilisation des requêtes arbre simples pour exprimer les contraintes). Par contre, puisque mes travaux sur web services sont plus récents, ils ont été présentés de façon plus succincte.

Dans ce cadre, cet annexe fournit *un* article pour illustrer chaque thème XML discuté dans ce mémoire. Par soucis d'équilibre et pour montrer l'évolution de nos recherches sur les services web, *tous* nos travaux publiés sont annexés.

1. L'article [BCH+07] décrit notre approche de validation incrémentale par rapport aux contraintes de schéma et aux contraintes de clés.
2. L'article [BCHS06b] présente notre méthode de correction des documents XML intégrée à la procédure de validation incrémentale.
3. L'article [dHM07] décrit l'approche `dGREC`, pour l'évolution des schémas ainsi que l'utilisation d'un validateur datalog.
4. Les articles [BCHM05, BHM06, BH08] montrent l'évolution de notre travail dans le domaine de services web. L'article [BH08] vient d'être accepté dans le *23rd ACM Symposium on Applied Computing - SAC - SIGAPP, Special Track on Web Technologies.*

[1] `http ://www.sir.blois.univ-tours.fr/ mirian/`